KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

Partial and dynamic FPGA reconfiguration for security applications

Jo Vliegen

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Technology

June 2014

Partial and dynamic FPGA reconfiguration for security applications

Jo VLIEGEN

Examination committee:
Prof. dr. ing. J. De Brabanter, chair
Prof. dr. ir. I. Verbauwhede, supervisor
Prof. dr. ir. N. Mentens, co-supervisor
Prof. dr. ir. Bart Preneel
Prof. dr. ir. Geert Deconinck
Prof. dr. ir. Dirk Stroobandt
(Universiteit Gent, Belgium)
Prof. dr. ir. Dirk Koch
(University of Manchester, United Kingdom)
Ir. Peter Grognard
(Septentrio n.v.)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Technology

June 2014

Uitgegeven in eigen beheer, Jo Vliegen, Kasteelpark Arenberg 10, bus 2452, B-3001 Leuven (Belgium)

ISBN 978-94-91857-02-7

D/2014/13.244/2

Preface

During the last four years, I was given an opportunity to work on FPGAs and to find ways to implement applications containing cryptographic components. Working in a research group with international fame is a privilege for which I'm sincerely grateful to my promoter Prof. Ingrid Verbauwhede and co-promoter Prof. Nele Mentens. My thanks also go to Prof. Bart Preneel and Prof. Geert Deconinck for being my assessors. These professors, with addition of Prof. Wim Dehaene, gave the best of themselves in the courses I attended in my first year and taught me valuable principles regarding digital design and cryptography. Thanks also go to every member of my examination committee.

Little research can be done without funding. For this, my appreciation goes to KU Leuven and KHLim for giving me their the opportunity to do my research. Special thanks for streamlining this go to dr. ir. Kris Henrioulle.

Next to the academical support, this doctorate could not have succeeded without the help of Péla Noë. She helped me a great deal with administration, which was not always straightforward when working at a campus outside Leuven. Also my current and former colleagues at KHLim often helped me with technical issues and by contributing in creating a pleasant work environment.

A list of friends, I would not dare to enumerate, are to be thanked. They assisted me with elevating my mood when needed and with proof-reading my thesis. Special thanks go to my parents, who always encouraged me to pursue the goals that excite and intrigue me, providing me with everything required to achieve these.

Finally, thanks go to the most important person in my life, Ellen. She encouraged me to seize this opportunity, stands by me in every challenge, and is a perfect mother to our daughter Lotte. It is to them that I dedicate this thesis.

Jo Vliegen
March 2014

Abstract

Today, FPGAs are still often used as prototyping devices. With the latest technologies to produce these devices, FPGAs are becoming larger and faster. With the correct development environments, FPGAs can also be used in end products to provide specialised features that are not always possible with ASIC chips such as in-field updates of hardware.

The contributions of this thesis start with implementations of cryptographic primitives. These implementations focus on using minimal resources and are uniformly wrapped to improve interconnectivity and interchangeability between components. Secondly, by using these primitives, we manage to realise the first single-chip, secure, and remote reconfiguration of an FPGA in the field. This is achieved by implementing a cryptographic protocol in combination with bitstream compression. The next contribution is made in realising a proof-of-concept implementation of a licensing scheme for hardware IP cores. We managed this by using a novel storage technique in combination with an academic tool. Our final contribution is in strengthening an existing distributed logging scheme, by moving the sensitive storage from server to FPGA. The reduction in communication overhead is achieved by using the cryptographic implementations on the FPGA to perform the calculations on the sensitive data.

When implementing applications that contain cryptographic components, a number of different fields of research come together. The most important research fields that meet in such an implementation focus on cryptology, hardware design, system-on-chip and design tools exploration. We tried to combine the evolutions in these different fields of research into our implementations.

Beknopte samenvatting

FPGA's worden vandaag nog altijd vaak gebruikt voor het maken van prototypes. Met de nieuwste technologieën om FPGA's te maken, worden deze devices almaar groter en sneller. Met de juiste ontwikkelomgevingen kunnen FPGA's ingezet worden in eindproducten om hierin functionaliteiten aan te bieden die met op maat gemaakte chips niet altijd mogelijk zijn zoals bijv. updates van hardware, na ingebruikstelling.

De bijdragen van dit proefschrift beginnen met de implementatie van cryptografische primitieven. Deze implementaties richten zich op het minimale gebruik van middelen en hebben een uniforme interface welke de interconnectiviteit en onderlinge uitwisselbaarheid verbeteren. Als tweede bijdrage slagen we er in om, met het gebruik van deze primitieven, de eerste single-chip, veilige FPGA-herconfiguratie van op een afstand uit te voeren. Dit wordt bereikt door het implementeren van een cryptografisch protocol in combinatie met bitstream-compressie. De volgende bijdrage situeert zich in het realiseren van een proof-of-concept implementatie van een licentiesysteem voor hardware-componenten. We zijn hierin geslaagd door middel van een nieuwe opslag-techniek en het gebruik van een academische tool. Onze laatste bijdrage is het versterken van een bestaand gedistribueerd logging concept door het verplaatsen van de opslag van gevoelige gegevens van de server naar FPGA. De vermindering in communicatie-overhead wordt mogelijk gemaakt door cryptografische implementaties op de FPGA te gebruiken voor de berekeningen op de gevoelige gegevens.

Wanneer toepassingen geïmplementeerd worden die cryptografische aspecten bevatten, komen uiteenlopende onderzoeksdomeinen samen. De belangrijkste domeinen hierbij, richten zich op cryptologie, hardware ontwerp, system-on-chip en ontwikkelomgeving verkenning. Wij hebben geprobeerd om de evoluties in deze verschillende onderzoeksdomeinen samen te brengen in onze implementaties.

Abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BisT	Board-is-Trusted
BRAM	Block Random Access Memory
CBC	Cipher Block Chaining mode (of operation)
CFB	Cipher FeedBack mode (of operation)
CLB	Configuration Logic Blocks
CPU	Central Processing Unit
CR	Compression Ratio
CTR	CounTeR mode (of operation)
DCM	Digital Clock Manager
DPA	Differential Power Analysis
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processing
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
ECP	Elliptic Curve Processor
ECPA	Elliptic Curve Point Addition
ECPD	Elliptic Curve Point Doubling
ECPM	Elliptic Curve Point Multiplication
FF	Flip-Flop
FisT	FPGA-is-Trusted
FPGA	Field Programmable Gate Array

FSM	Finite State Machine
GF	Galois Field
HMAC	Keyed-Hash Message Authentication Code
HO-DPA	Higher-Order Differential Power Attacks
ICAP	Internal Configuration Access Port
IP	Intellectual Property
IP	Internet Protocol
KAP	Key Agreement Protocol
KDF	Key Derivation Function
LUT	Look-Up Table
LZW	Lempel–Ziv–Welch
MAC	Media Access Control
MAC	Message Authentication Code
MAS	Modular Adder Subtractor
MM	Modular Multiplier
NIST	National Institute of Standards and Technology
NVM	Non-Volatile Memory
OFB	Output FeedBack mode (of operation)
PCB	Printed Circuit Board
PLB	Processor Local Bus
PUF	Physically Unclonable Function
RLE	Run-Length Encoding
SHA	Standard Hash Algorithm
SPA	Simple Power Analysis
SRAM	Static Random-Access Memory
STS	Station-To-Station
TA	Timing Attack
TTP	Trusted Third Party
UDP	User Datagram Protocol

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter situates the research of this thesis. After giving a motivating example, it introduces the Field Programmable Gate Array (FPGA), for which this research is tailored. As the contributions of this research are focused on cryptographic implementations and their applications, it subsequently provides the reader with a cryptologic background. Finally it makes a summary of the contributions of this work and presents an overview of the upcoming chapters.

1.1 Motivation

1.1.1 Motivating example

In August 1955, Life Magazine published an article "Throwaway Living" in which it first introduced the term *throwaway society*. Nowadays the term is known by everyone and covers the excessive production of disposable items. With the fast pace of technological evolution, this concept also applies to electronic devices. Instead of forcing customers to upgrade their electronic devices, updating their currently owned devices might be a more practical, ecological, and economical solution.

Nowadays, applying this concept to software is something which everyone is familiar with. A look at the brief history of game consoles illustrates this.

In Figure 1.1 a console is shown for the popular game 'Donkey Kong Jr.' which was released in 1982. On this mobile console only the 'Donkey Kong Jr.' game could be played. Four years later the game was available for the Nintendo

Figure 1.1: The wide-screen Donkey Kong Jr. console

Figure 1.2: The Nintendo Entertainment System

Entertainment System, shown in Figure 1.2. This 8-bit video console could be used to play various games that could be loaded through a cartridge or a 'Game Pak', a picture of which is shown in Figure 1.3. This shift allowed users to own a single device on which many games could be played by replacing the software that ran on the microprocessor. The Game Pak contained a printed circuit board which mainly held a ROM with the game's code. Updating the running program, or at least a part of it, could easily be done by physically switching the Game Pak.

With the growth of gaming on personal computers, large groups of users switched platform, but the external software storage remained. Games were to be bought and physically connected to the platform. Approximately seventeen years after the release of the Nintendo Entertainment System, the Steam platform was released in 2003 by Valve Corporation (Figure 1.4). This platform offers, amongst other core features, a digital distribution channel. Steam users can simply buy and download games from the Internet to the computer, thus avoiding the physical intervention of external memory.

This evolution in software could be followed by hardware: moving from a device with a sole purpose to a device with a programmable processor, that can be altered to serve multiple purposes or to be updated, without the physical intervention of a technician.

1.1.2 Reconfigurable hardware

Devices of which the hardware can be altered in the field, could prolong the life-cycle of a product or could combine the core tasks of multiple products. Achieving such goals could contribute to converting the throwaway society into a more ecological reconfig-and-replay society. In order to achieve this, there is

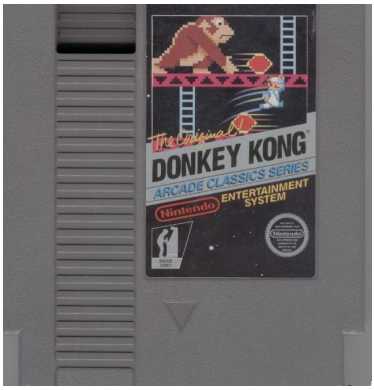

Figure 1.3: A Donkey Kong Game Pak for the Nintendo Entertainment System

Figure 1.4: The logo of Steam, released by Valve Corporation in 2003

a need for hardware which can be easily updated.

Apart from prolonging the life span of a product, a more dynamic use of the reconfigurable property of reconfigurable devices can broaden the type of applications which can benefit from reconfigurable hardware. For example, although using dedicated hardware as co-processors is not ground-breaking, being able to update them is not so common. This would be a welcome feature for co-processors in fields that evolute at a fast pace, such as cryptography. Another example is the use of Intellectual Property (IP) cores. With reconfigurable hardware, the flexible usage of IP cores becomes possible. This flexibility can be reflected in: obtaining IP cores, tracking IP core copies, or even renting IP cores for the usage of certain amount of time or number of uses.

With the setting of devices that contain reconfigurable hardware already deployed in the field, securing the communication with such a device becomes a necessity.

Academic research in fields as hardware reconfiguration, cryptography, IP core management, and hardware implementations goes very broad and very deep. More scarce is the research focusing on combining these different fields into a single, viable solution. This thesis tries to fill this gap by tackling three issues that require configurable hardware security, namely secure logging, secure remote configuration and secure IP core licensing.

To achieve reconfigurable devices in the future, this research uses a Field Programmable Gate Array (FPGA) as hardware platform. The FPGA is thoroughly discussed in Section 1.2. To provide the reader with a basic

background on cryptology, Section 1.3 introduces the most relevant concepts, primitives, mathematical principles and physical attacks. Finally this chapter summarises the contributions of this dissertation in Section 1.4, Section 1.5 gives an overview of the remainder of this manuscript, and Section 1.6 summarises the conclusions of this chapter.

1.2 Field Programmable Gate Array

The name Field Programmable Gate Array was used for the first time in 1985. It was developed by Ross Freeman and Bernard Vonderschmitt who founded Xilinx in 1984, together with Jim Barnett. At that time, the name quite accurately described what the chip was: a gate array that could be programmed in the field, and of which the gate connections are programmable as well. Almost twenty years later, an FPGA contains huge amounts of programmable gates, together with a wide variety of dedicated cores.

1.2.1 FPGA technology

An FPGA consists for a very large part out of reconfigurable blocks for the implementation of logical functions and switch matrices for routing. To impose a certain behaviour on the FPGA, the reconfigurable blocks have to be configured. The file that contains this configuration is referred to as a bitstream. The technology to store the configuration determines the technology of the FPGA. Analysis of the current FPGA market shows the three most common technologies are:

SRAM-based FPGAs store the configuration in static random-access memory (SRAM) cells. These FPGAs need to be reconfigured at each start-up due to the volatile nature of the memory. The static power consumption of an SRAM-based FPGA is relatively high.

Flash-based FPGAs store the configuration in an EEPROM-based Flash memory. Because Flash memory is non-volatile, these FPGAs keep their configuration at start-up, but are less appropriate for a large amount of reconfigurations. The Flash-based FPGA does not require a permanent power supply, which lowers the static power consumption.

Antifuse-based FPGAs store the configuration by making permanent connections. These FPGAs cannot be reconfigured, but allow higher clock-rates. The Antifuse-based FPGA consumes the least static power

since the low impedance metal contact of an Antifuse connection causes less power loss than a switched-on transistor.

Apart from the maximum number of reconfigurations, static power consumption is a major difference between these different FPGA technologies. Additionally, it is pointed out that on power-up the SRAM-based FPGA will draw an inrush current, because the N-MOS and P-MOS transistors are both in a low impedance state, which introduces a delay before the SRAM can be programmed. This delay is non-existing in Antifuse or Flash-based FPGAs, which are 'live upon start-up'.

Figure 1.5 shows the market shares of the three largest FPGA vendors in 2013, based on their yearly revenue. As can be seen in Table 1.1, this roughly indicates that over 90% of all FPGAs are SRAM based.

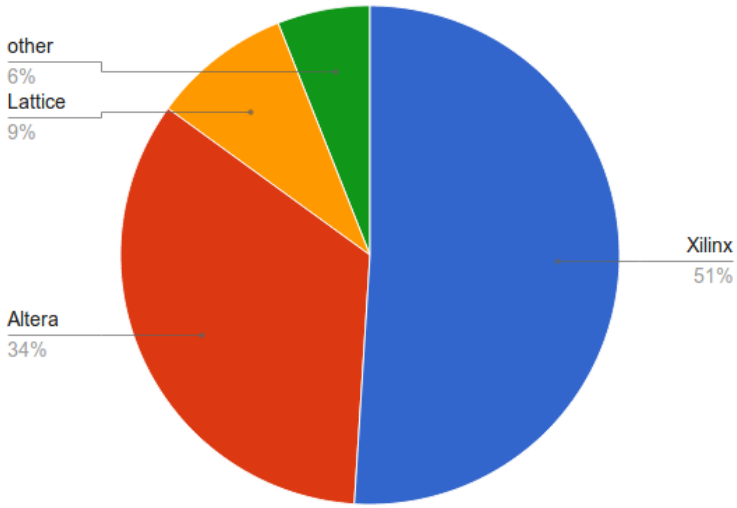

Figure 1.5: FPGA market shares 2013

Table 1.1: Overview of the available FPGAs of the most important vendors based on the technology

	Xilinx	Altera	Lattice	Microsemi
SRAM	✓	✓	✓	✓
Flash	✗	✗	✗	✓
Antifuse	✗	✗	✗	✓

Until recently Xilinx offered two main lines of FPGAs: the Virtex FPGAs and the Spartan FPGAs. The former were the high-end devices, while the latter were the cheaper low-end devices. In the most recent Xilinx FPGA family, the 28 nm technology based 7-series, the Spartan-line FPGA is replaced by two similar new lines: the Kintex FPGAs and the Artix FPGAs of which the latter is a more power-friendly sibling.

In this work, we use SRAM FPGAs because of their ubiquity. Moreover, dynamic and partial reconfiguration is only supported in SRAM-based FPGAs. Because partial reconfiguration is important for this work, all targeted devices in this work are Xilinx products. This vendor introduced the partial reconfiguration technique and has been supporting it for more than a decade.

1.2.2 The configuration of an FPGA

As mentioned above, the FPGA gets its configuration through a bitstream. This section describes how a bitstream is generated for Xilinx FPGAs and how this bitstream configures the FPGA.

Bitstream generation

The first step in the creation of a bitstream is to describe the design in a Hardware Description Language such as: VHDL, Verilog or SystemC, or through schematic entry. The design has to go through the synthesis tool to be transformed into a netlist (a *.ngc* file). This netlist then gets combined with any other cores in the Build step, which results in a *.ngd* file. This *.ngd* file contains a logical description in terms of logic elements such as AND gates, flip-flops, and similar gates. Thereafter, this *.ngd* file gets mapped on the hardware primitives, which are the building blocks of the reconfigurable fabric of the targeted FPGA, resulting in a *.ncd* file. The subsequent step is to place and route (PAR) the mapped design. Placing is determining where each primitive from the mapping phase is placed on the chip, while routing tries to make every required connection through the routing lines on the chip. These steps result in a routed *.ncd* file which, finally, can be used to generate a bitstream for the FPGA. These different processing steps and the corresponding file extensions are shown in Figure 1.6.

Although the FPGA vendor provides the system designer with the required tools to run each of the above mentioned steps, other commercially available tools exist to perform the synthesis step. Although these tools achieve better results, the license price is also of a different magnitude.

Figure 1.6: The processing steps with the corresponding file extensions of the bitstream generation process

FPGA configuration

By loading the configuration memory with a bitstream, the behaviour of the reconfigurable fabric is determined. Commonly an FPGA is configured in its entirety, but partial reconfiguration is a technique that allows the reconfiguration of a certain partition of an FPGA. A bitstream that contains the configuration of a partition of the FPGA is referred to as a partial bitstream. The partition that stays unchanged is referred to as the 'static partition' while the one or more other partitions are referred as 'reconfigurable partition(s)'. If the operation of the static partition continues uninterruptedly during the reconfiguration this is referred to as 'dynamic' partial reconfiguration. The three different approaches to reconfigure an FPGA are shown in Figure 1.7. The possibility exists to use encrypted bitstreams to configure FPGAs without revealing the bitstream. An on-chip decryption core is present to perform bitstream decryption, but up to the latest Xilinx-series [82, 83] this feature is not available for partial bitstreams. Moreover, this decryption core is not available for any purpose other than bitstream decryption.

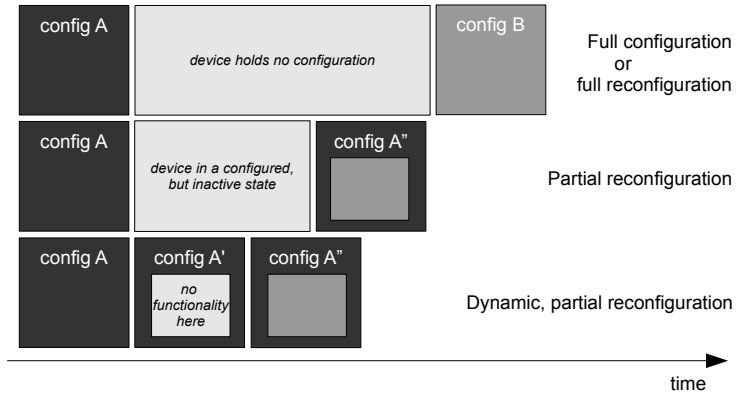

Figure 1.7: Three different approaches to reconfigure an FPGA

Table 1.4, at page 25, summarises which recent Xilinx FPGA families offer certain features related to secure (re-)configuration. This table shows that data authentication is only available in the Virtex-6 and the 7-series. The purpose of

this authentication is twofold. The first purpose is to provide a mechanism that verifies that the FPGA configuration is exactly the same as the intended FPGA configuration. This detects tampering. The second purpose is to guarantee that this configuration originates from a trusted origin. Data authentication and other terminology are further explained in Section 1.3.1.

1.2.3 The anatomy of the targeted devices

This research is targeted at two different devices: the XC5VFX70T and the XCS6LX45. The XC5VFX70T is a Virtex-5 family member, which targets high-performance embedded systems with advanced serial connectivity [78], and the Spartan-6 XCS6LX45, which targets high-volume applications requiring leading system integration capabilities with the lowest total cost [79]. Both FPGAs' Configuration Logic Blocks (CLB), shown in Figure 1.8 and Figure 1.9, visualise how the switch matrices are connected to a single CLB. For both devices, each CLB holds two slices. The main difference between these CLBs is the CIN and COUT connection through the right slice, which is only available in the Virtex-5 family. These dedicated connections are to route carry logic which is frequently used in arithmetic operations.

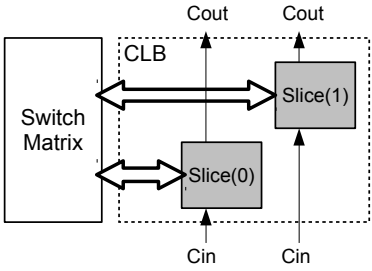

Figure 1.8: The CLB of a Virtex-5 FPGA

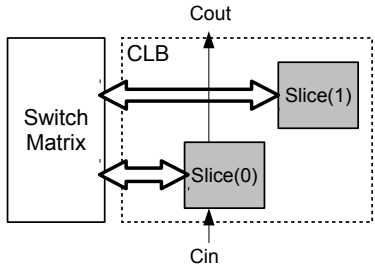

Figure 1.9: The CLB of a Spartan-6 FPGA

Although the content of the CLBs shown in Figure 1.8 and Figure 1.9 look very similar, the content of their slices differs significantly, as summarised in Table 1.3. This table gives the different types of available slices for each targeted device. The Slice-L and Slice-M both offer support for arithmetic circuits using carry chains whereof Slice-M additionally provides distributed memory functionality. The more light-weight Slice-X is only available in the Spartan-6 devices. Although the Spartan devices have larger look-up tables (LUT) and more flip-flops (FF) in their slices, the high-end Virtex devices have much more slices.

The LUT is the smallest building block to provide reconfigurable 'gates'. An n-input LUT can emulate every possible function with n inputs. This means that each LUT can be configured to 2^{2^n} different functions.

Table 1.2: The truth table for any 2-input function, where $\cdot$ is a logical AND, $+$ is a logical OR, $\oplus$ is an exclusive OR, and an overlined character represents an inverted value.

A	B	F1	F2	F3	F4	F5	F6	F7	F8
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1
		'0'	$A \cdot B$	$A \cdot \overline{B}$	A	$\overline{A} \cdot B$	B	$A \oplus B$	$A + B$
		F9	F10	F11	F12	F13	F14	F15	F16
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1
		$\overline{A + B}$	$\overline{A \oplus B}$	$\overline{B}$	$A + \overline{B}$	$\overline{A}$	$\overline{A} + B$	$\overline{A \cdot B}$	'1'

As mentioned above, the FPGAs of today also contain a wide variety of dedicated cores on the FPGA die[1]. Although the details may vary between different vendors' products, most of these cores can be found somewhere in each vendors' offering. The most relevant cores are explained here and are also taken up in Table 1.3.

BRAM The Block Random Access Memory (BRAM) cores are dedicated memory cores. These cores can be configured to have different length and depth configurations. A single 18 Kibit BRAM can, for example, be configured as a memory offering 1024 addresses which each can store an 18-bit value, while it can also be configured as a memory with 8192 locations for 2-bit values. The targeted devices both provide the feature to have a dual-ported BRAM which offers two independent read-and-write ports on the same memory space. BRAMs can also be configured as FIFOs.

[1] A die is a small piece of semiconducting material on which a circuit is implemented.

DSP slices The Digital Signal Processing (DSP) slices are able to implement certain mathematical functions very fast because of their dedicated nature. These functions can be used to configure an adder, a multiplier, a 48-bit logic operation block or other mathematical or logical function. As an example of a simplified operation, the functions can be fit to: $O = (Z \pm (X + Y + C_{in})\ OR\ (-Z + (X + Y + C_{in})))$, to implement a three input adder.

DCM The Digital Clock Manager (DCM) is a core that provides the possibility to change between or provide multiple clock domains. The use of a DCM minimises clock skew and duty cycle distortion.

Switch matrix A switch matrix provides programmable routing in the FPGA. Horizontal and vertical wires cross each other in a switch matrix and, depending on the SRAM configuration cell of the switch matrix, may be interconnected and/or routed to a CLB. Thus, these switch matrices interconnect the CLBs.

Table 1.3: Differences in slice content between the XC5VFX70T and the XCS6LX45

	Virtex-5 FX70T		Spartan-6 LX45		
	L	M	X	L	M
share	±75%[63]	∓25%[63]	50%	25%	25%
LUT	6I	6I	6I	6I	6I
#FF	4	4	8	8	8
wide MUXes	✓	✓	✗	✓	✓
carry logic	✓	✓	✗	✓	✓
distributed RAM	✗	✓	✗	✗	✓
shift registers	✗	✓	✗	✗	✓
#slices	11 200		6 822		
#LUT	44 800		54 576		
#BRAM	148 × 36 Kibit		116 × 18 Kibit		
#DSP[1]	128		58		
#DCM	12		8		

[1]DSP48Es for Virtex-5 and DSP48A1s for Spartan-6
X slice: basic Spartan-6 slice
L slice: a slice containing a carry structure and multipliers
M slice: an L slice with distributed RAM and shift registers

1.3 A cryptologic backpack

This section presents the reader with general background on what cryptology is and explains important concepts in Section 1.3.1. The most relevant cryptographic primitives are explained more thoroughly in Section 1.3.2. To implement certain primitives, mathematical systems are used for which Section 1.3.3 provides the relevant algebraic background. Finally, Section 1.3.4 introduces physical attacks.

1.3.1 Cryptologic background

Cryptology is the study of secure communications and it covers both cryptography and cryptanalysis. The former is the study on schemes to allow secure communications, while the latter tries to break or avoid these schemes.

Cryptography relies on security primitives. In [45], Menezes et al. report three main categories of security primitives: unkeyed primitives, public-key primitives and symmetric-key primitives. Figure 1.10 shows which primitives belong to these categories. For the sake of completeness, it is mentioned that symmetric-key ciphers can be divided again into block ciphers and stream ciphers. Before explaining these primitives, some important cryptographic concepts are defined first. For more complete background information the reader can consult [43, 45].

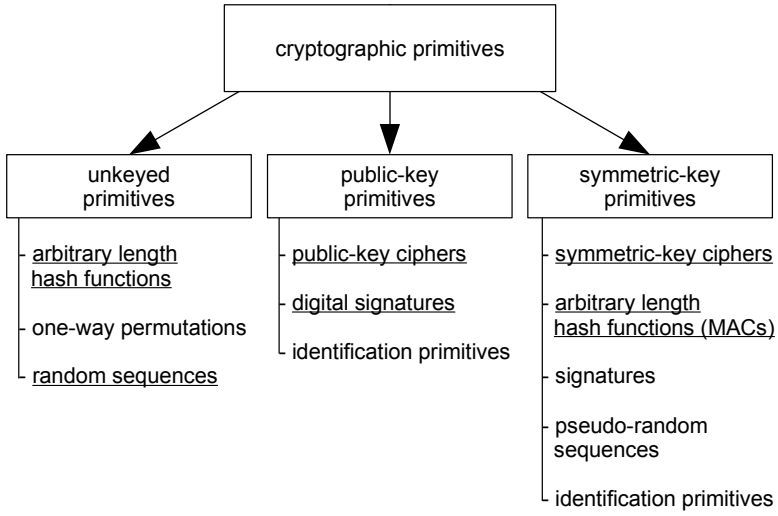

Figure 1.10: The cryptographic primitives as discussed in [45]

A number of important cryptologic concepts are described using Figure 1.11. It is assumed that Alice and Bob want to have secure communication over an unsecure channel, to which their adversary, Eve, has access. Six common cryptographic goals are:

data confidentiality encompasses that the messages sent between Alice and Bob, cannot be understood by Eve.

entity authentication or **identification** encompasses that Bob can be sure the other entity is Alice and not Eve, possibly pretending to be Alice.

data-origin authentication encompasses that Bob can be sure the message is sent by Alice.

data integrity encompasses that Bob can be sure the message is not tampered with.

data authentication encompasses both data-origin authentication and data integrity.

non-repudiation encompasses in one direction that Alice cannot deny having sent a certain message and in the other direction that Bob cannot deny having received a certain message.

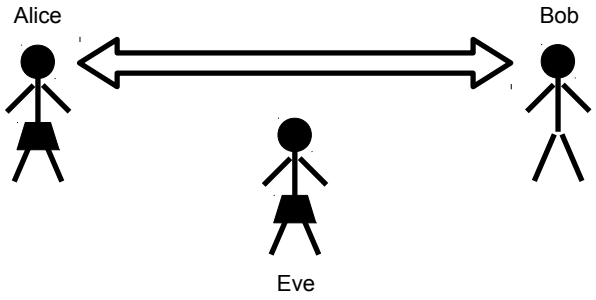

Figure 1.11: Entities

To achieve these goals, a cryptographic protocol is required. Such a protocol is a well-defined procedure in which Alice and/or Bob send messages to each other after performing operations on these messages. Protocols are studied thoroughly in cryptography and exist in many different flavours. Often, these protocols contain 'encryption'. To be clear on what is covered by this and other terms throughout this dissertation, some definitions are given.

Encryption is the process in which a message is transformed in some incomprehensible form. A message plainly readable to anyone is referred to as the **plain text**, while an encrypted message is referred to as the **cipher text**. The inverse transformation is **decryption**.

When both Alice and Bob use the same cryptographic key for the encryption and decryption of a message, they use **symmetric-key cryptography**. Although symmetric-key cryptography provides a solution for a number of cryptographic concepts, securely establishing a situation where only Alice and Bob possess the symmetric key is a huge challenge for large scale systems. **Public-key cryptography** can be used to overcome this challenge by using key pairs for every communicating entity. One key is kept private while the other key can be made public.

1.3.2 Cryptographic primitives

The underlined primitives in Figure 1.10 are frequently used in this dissertation. These primitives are elaborated on here.

A hash function

This unkeyed primitive is an algorithm that maps an input of an arbitrary length to an output of a fixed length, called the 'hash value'. Hash functions used in cryptographic applications should have the following properties:

collision resistance ensures that it is hard to find any two distinct inputs with the same hash.

pre-image resistance ensures that it is hard, given a hash value h, to find an input x which has h as a hash.

second pre-image resistance ensures that it is hard, given an input x, to find a second input y such that the hash of x is equal to the hash of y.

A hash-function is often used to generate a 'message fingerprint' of a message.

Random sequences

Random sequences are essential in cryptographic schemes. It should be infeasible to predict the outcome of the random sequence. There are two different methods

for generating random numbers: true random number generators and pseudo-random number generators.

True random number generators start from a physical phenomenon which is measured. It might be possible that taking these measurements introduces a bias to the generated number. When this occurs it should be compensated for with appropriate post-processing. Pseudo-random number generators follow an algorithm which is initialised with a symmetric key or random seed. The generator then produces seemingly random numbers.

Random numbers are, as an example, used in the generation of keys or for the initialisation of a system.

A cipher

Figure 1.12 visualises a cipher which is used to encrypt (decrypt) messages. The input of the cipher comes in from the left, while a key comes in from below. The output will appear on the right side of the cipher. When the plain text (cipher text) is input, the cipher performs encryption (decryption). Whether the key is a private or symmetric key, determines if the cipher is referred to as a public-key cipher or as a symmetric-key cipher, respectively.

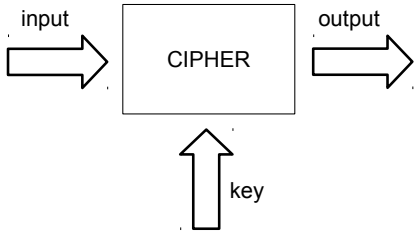

Figure 1.12: Cipher

As mentioned before, symmetric-key ciphers can be subdivided into block ciphers and stream ciphers. Block ciphers take a block with a certain number of plain text symbols to encrypt in one iteration to produce a block of cipher text, while stream ciphers operate on a stream of single plain text symbols.

A message authentication code

As the name states, a message authentication code (MAC) is a string which is used to authenticate a message. More strictly, the message is input for a complex function together with a secret key. The result of the function is

referred to as the MAC value and is typically combined with the message to protect its authenticity.

A digital signature

A commonly known signature is a mechanism to bind an entity to a specific piece of information. The digital equivalent of a signature is a digital signature. Although both symmetric-key and public-key primitives can provide digital signatures, public-key primitives have an advantage over symmetric-key primitives. A public-key primitive would generate a signature on the message with the private key, which everybody can verify with the according public key; whereas with a symmetric-key primitive both the signer and the verifier use the same key. The non-repudiation concept can be achieved through the former, because there is only a single entity that can generate signatures.

1.3.3 Algebraic background

Because cryptographic primitives often make use of calculations in a finite field, this section recapitulates the relevant algebraic structures: group, ring and field.

A 'Group' $< A, \cdot >$ is an algebraic structure that consists of a set of elements A and an operation $\cdot$. For $< A, \cdot >$ to be a group, the four group axioms $G1$ - $G4$ have to hold.

$G1:$ A has closure under $\cdot$
$\forall x, y \in A : \exists z \in A; x \cdot y = z$

$G2:$ $\cdot$ has the associative property
$\forall x, y, z \in A : x \cdot (y \cdot z) = (x \cdot y) \cdot z$

$G3:$ there is an (two-sided) identity (or neutral) element for $\cdot$ in A
$\exists! e \in A; \forall x \in A : x \cdot e = e \cdot x = x$

$G4:$ there is an (two-sided) inverse element for $\cdot$ in A
$\forall x \in A : \exists! x^{-1} \in A : x \cdot x^{-1} = x^{-1} \cdot x = e$

If a group has a finite number of elements in its set A, it is referred to as finite group where the number of elements is called the order of A. If the additional axiom $G5$ is also met in a group $< A, \cdot >$, this group is called an Abelian (or commutative) group. The most common example of an Abelian group is the set of integers under the addition operation.

$G5:$ $\cdot$ has the commutative property
$\forall x, y \in A : x \cdot y = y \cdot x$

A 'Ring' $< A, \cdot, + >$ is an algebraic structure which consists of a set of elements A and two operations $\cdot$ and $+$. For $< A, \cdot, + >$ to be a ring, axioms $R1$ - $R4$ have to hold.

$R1:$ $< A, + >$ is an Abelian group
with '0' as neutral element and '$-a$' as inverse element

$R2:$ A has closure under $\cdot$

$R3:$ $\cdot$ has the associative property

$R4:$ $\cdot$ is distributive over the Abelian group operation $(+)$
$\forall x, y, z \in A : x \cdot (y + z) = x \cdot y + x \cdot z$
and
$\forall x, y, z \in A : (x + y) \cdot z = x \cdot z + y \cdot z$

If the additional axiom $R5$ is also met, the ring $< A, \cdot, + >$ is called a commutative ring. If axiom $R6$ holds in addition to $R1$ - $R4$, the ring $< A, \cdot, + >$ is called a 'ring with multiplicative identity'. When axioms $R6$ and $R7$ hold in addition to $R1$ - $R4$, the ring $< A, \cdot, + >$ is called a 'skew field'. Finally, when the axioms $R1$ - $R7$ hold, the algebraic structure is called a 'field'.

$R5:$ $\cdot$ has the commutative property

$R6:$ there is an (two-sided) identity element for $\cdot$ in A

$R7:$ there is an (two-sided) inverse element for $\cdot$ in $A \setminus \{0\}$

Moreover, if a field has a finite set of elements, it is referred to as a finite field or a Galois Field (GF). A common example of a field is the set of real numbers, with operations addition and multiplication.

Groups constructed using an elliptic curve

Koblitz [33] and Miller [48] introduced the use of elliptic curves for constructing cryptographic primitives. These curves are defined over a finite field $GF(p)$ by Equation (1.1), where a and b are constants in $GF(p)$ that satisfy Equation (1.2).

$$y^2 = x^3 + ax + b \tag{1.1}$$

$$4a^3 + 27b^2 \not\equiv 0. \tag{1.2}$$

The set of elements of the finite field are the points on the curve. To meet requirement $G3$, an additional point $\mathcal{O}$ is required. This point is referred to as 'the point at infinity' and is formulated as $\mathcal{O} = (x, \infty)$. The complete set of elements is hence defined by Equation (1.3).

$$E = \{P = (x, y) \mid x, y \in GF(p) \; solved \; from \; (1.1)\} \cup \{\mathcal{O}\}. \tag{1.3}$$

The group operation is the elliptic curve point addition (ECPA). The ECPA calculates the sum of two points according to Equation (1.4).

$$P_3 = P_1 + P_2 = (x_1, y_1) + (x_2, y_2) = (x_3, y_3) \tag{1.4}$$

with

$$x_3 = (\lambda^2 - x_1 - x_2)$$

$$y_3 = ((x_1 - x_3) \cdot \lambda - y_1)$$

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}.$$

The Elliptic Curve Point Multiplication (ECPM) is the multiplication of a scalar k with a point $P \in E$. It can be calculated by subsequently adding the same point $k - 1$ times. A more efficient approach is by using the 'double-and-add' method which is shown in Algorithm 1.

ALGORITHM 1: ECPM using double-and-add

Input: point P, non-negative scalar $k = (1k_{l-2}...k_1k_0)$
Output: point Q, with $Q = kP$
$Q \leftarrow P$;
for *index in* $[(l-2)..0]$ **do**
    $Q \leftarrow 2Q$;
    if $k_{index} = 1$ **then**
        $Q \leftarrow Q + P$;
    end
end

Montgomery presented a slightly different method in [50] which is shown in Algorithm 2. Although this method is less efficient in terms of number of

ALGORITHM 2: ECPM using a Montgomery ladder

Input: point P, non-negative scalar $k = (1k_{l-2}...k_1k_0)$
Output: point Q, with $Q = kP$
$Q \leftarrow P$;
$S \leftarrow 2P$;
for *index in* $[(l-2)..0]$ **do**
    if $k_{index} = 1$ **then**
        $Q \leftarrow Q + S$;
        $S \leftarrow 2S$;
    else
        $S \leftarrow Q + S$;
        $Q \leftarrow 2Q$;
    end
end

operations, it results in a scalar-value independent run-time. The benefits from this will be explained in Section 1.3.4.

Both in Algorithm 1 and Algorithm 2 an elliptic curve point doubling (ECPD) is used, which is a special case of a point addition. An ECPD can be calculated according to Equation (1.5).

$$P_3 = 2P_1 = 2(x_1, y_1) = (x_3, y_3) \tag{1.5}$$

with

$$x_3 = (\lambda^2 - 2 \cdot x_1)$$

$$y_3 = ((x_1 - x_3) \cdot \lambda - y_1)$$

$$\lambda = \frac{3 \cdot x_1^2 + a}{2 \cdot y_1}.$$

For a graphical representation of these operations and for a more in depth explanation, [43] can be consulted.

NIST recommended curves

Elliptic curve computations consist of operations in the underlying finite field. The most widely used fields are binary fields and prime fields, denoted by $GF(2^n)$ and $GF(p)$, respectively. Because the hardware implementation of binary field operations results in carry-free logic, these fields are optimal for use

in hardware in terms of speed and area. On the other hand, some cryptographic algorithms also require modular operations over $GF(p)$ next to the elliptic curve, which makes $GF(p)$ more attractive in terms of resource sharing.

In [60], NIST[2] recommends the use of 15 different curves for elliptic curve cryptography. For prime fields, there are five curves where the size of the prime number is 192, 224, 256, 384, and 521 bits. For binary fields, there are five fields with n equal to 163, 233, 283, 409, and 571. For every binary field, NIST recommends one elliptic curve and one Koblitz curve[3].

1.3.4 Physical attacks

Cryptanalysis can coarsely be divided into three groups: attacks on algorithms, brute-force attacks and physical attacks. The first tries to exploit weaknesses in algorithms and/or protocols. Mathematical proof together with large exposure to possible threats could be an indication a certain algorithm or protocol is not vulnerable to these types of attacks.

The second group of attacks tries to find a cryptographic key by systematically trying every possibility for the key. This theoretically might require to check every possible key in the search space. These attacks could be countered by designing a large enough search space. This can be done by increasing the alphabets and the lengths of the keys.

The third group of attacks tries to break a cryptographic system by gathering additional information from an implementation of a cryptographic system. Side-channel attacks can target different sources, such as power consumption, electromagnetic radiation, acoustic information, or can gather information by introducing faults to the implementation. In this dissertation, only side-channel attacks based on power consumption variations are considered.

This section briefly introduces timing analysis (TA), simple power analysis (SPA), and differential power analysis (DPA).

TA

A timing analysis tries to gather information by accurately measuring the duration of specific computations. Its first occurrence was in 1996 [38], where Kocher published timing attacks on various implementations.

[2]National Institute of Standards and Technology (NIST) is an agency of the United States Departement of Commerce

[3]Koblitz suggested to use a generalisation of elliptic curves for cryptography in [34]. This is referred to as HyperElliptic Curve Cryptography (HECC).

SPA

The simple power analysis attacks involve 'visual' inspection of power traces, representing power usage of a device over time, and can be used to distinguish large features. Examples of such features are: the number of rounds in AES [55] and, at a high magnification, to distinguish a multiplication from a squaring operation. Applying this latter example on an ECPM, leaks the scalar on a double-and-add implementation of Algorithm 1, because the ECPA only occurs for a scalar bit '1'. An implementation of Algorithm 2 results in a more time constant implementation.

DPA

Differential power analysis attacks were introduced by Kocher et al. [39]. They combine statistical analysis and error correction techniques to extract secret information. DPA attacks consists of two stages: data collection and data analysis. In the data collection stage a certain operation, executing a specific task with specific parameters, is to be measured multiple times, while the data analysis stage processes these measurements.

High-order differential power attacks (HO-DPA) are an advanced form of DPA taking into account multiple data sources for the data collection phase. HO-DPA was originally proposed by Messerges [47] and Chari et al. [9].

1.4 Summary of contributions

In most applications, cryptography is regarded as overhead with respect to the main application. Hence, for the applications considered in this dissertation, the area overhead is the most critical. Therefore, every implementation of cryptographic components in this thesis is focused towards minimal area. Keeping the required resources to a minimum, leaves the maximum amount of resources for the main application, when both parts are implemented on a single FPGA. A contribution is made in wrapping these cryptographic components in a uniform wrapper to improve interconnectivity and interchangeability between components.

If we want to contribute to having hardware follow the evolutions that software has been through, a first required step is the secure and remote reconfiguration of an FPGA deployed in the field. Our contribution in achieving this requirement is the first implementation of a secure and remote reconfiguration of an FPGA.

Moreover, we achieved to make this implementation as a single-chip solution by implementing a cryptographic protocol in combination with bitstream compression.

We also made a contribution in realising a licensing scheme for hardware IP cores. To achieve this, we used a novel storage technique and an academic tool to overcome the challenges in implementing a proof-of-concept implementation of an existing and sound licensing scheme.

Our final contribution is in strengthening an existing distributed logging scheme, by moving the storage of sensitive data from server to hardware. Without precautions, this would result in a substantial communication overhead, during which the sensitive data may be exposed. The reduction in communication overhead and the avoidance of exposition is achieved by adding implementations of cryptographic components on the FPGA, in order to perform the calculations on the hardware.

When implementing applications that contain cryptographic components, a number of different fields of research come together. The most important research fields that meet in such an implementation are cryptology, hardware design, system-on-chip, and design tools exploration. We tried to combine the evolutions in these different fields of research into our implementations of the applications.

In the remainder of this section, a summary of the published contributions that resulted from this thesis is given. The publications are grouped per chapter of this dissertation and contain a single line description that describes the actual contribution.

Chapter 2

The design and discussion of the elliptic curve processor was presented at the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), entitled: "*A compact FPGA-based architecture for elliptic curve cryptography over prime fields*", by Vliegen, Mentens, Genoe, Braeken, Kubera, Touhafi, and Verbauwhede. This paper thoroughly describes the processor, which we built from scratch. At the moment of publication, this implementation was the smallest implementation of its kind, without having an effect on the duration of its calculation. My contribution is the design and the implementation of the architecture together with the writing of the paper.

Chapter 3

A paper, focused on the hardware strengthening of the distributed logging scheme, was published at the 15th Euromicro Conference on Digital System Design (DSD) under the trivial title "*Hardware strengthening a Distributed Logging Scheme*", by Vliegen, Wouters, Grahn, and Pulls. The focus in this publication was on the proof-of-concept of the hardware strengthening. My contribution is the design and the implementation of the architecture of the strengthening together with the writing of half the paper.

A more elaborated publication of the overall scheme and the hardware strengthening is published as a Technical report of the Karlstad University, in Sweden: "*Distributed Privacy-Preserving Log Trails*", by Pulls, Wouters, Grahn, and Vliegen. My contribution is a chapter in this report, that elaborates on the hardware strengthening.

Chapter 4

The idea of remote reconfiguration of FPGAs is presented at the Dagstuhl seminar 'Dynamically Reconfigurable Architectures' by Mentens, Vliegen, Braeken, Touhafi, Verbauwhede, and Wouters, entitled: "Secure remote reconfiguration of FPGAs". This was an overview presentation that contains results of my work.

The first implementation of this idea was presented at the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoc), by Braeken, Genoe, Kubera, Mentens, Touhafi, Verbauwhede, Verbelen, Vliegen, and Wouters. This work achieves remote FPGA reconfiguration by sending full bitstreams to the FPGA. These bitstreams are stored in Flash memory on the printed circuit board (PCB). My contribution is the design and implementation of the communication core and writing of half the paper.

The introduction of partial reconfiguration in remote FPGA updates is accepted for publication in the ACM Transactions on Reconfigurable Technology and Systems, entitled: "Secure, remote, dynamic reconfiguration of FPGAs", by Vliegen, Mentens, and Verbauwhede. My contribution is the design and implementation described in this article together with writing it.

A reinforcement on this work is to restrict the trusted zone to be only the FPGA chip. A publication of this Single-chip solution was presented by Vliegen, Mentens, Verbauwhede at the 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig), under the title: "A Single-chip Solution for the Secure Remote Configuration of FPGAs using Bitstream Compression". My

contribution is the design and implementation described in this paper together with writing it.

Chapter 5

The results of the chapter that handles on the proof-of-concept implementation of the licensing scheme is submitted to the Journal of Cryptographic Engineering. This submitted article is entitled: "Practical Feasibility Evaluation and Improvement of a Pay-per-Use Licensing Scheme for Hardware IP Cores in FPGAs", by Vliegen, Mentens, Koch, Schellekens, and Verbauwhede. My contribution is the design and implementation of the static system, the generation of the partial bitstreams, and the writing of the article.

Not discussed in this thesis

The first publication which is not discussed in this thesis, is: "Secure FPGA technologies and techniques", by Braeken, Kubera, Trouillez, Touhaffi, Vliegen and Mentens. It was presented at the 19th International Conference on Field Programmable Logic and Applications (FPL) and presents the results of a literature study on the different types of FPGAs and their suitability to cryptographic implementations. My contribution is limited to revision of the paper.

An article in the ARPN Journal of Systems and Software, by Verbelen, Braeken, Kubera, Mentens, Touhafi, and Vliegen, discusses the server architecture of a system that controls and manages the remote updates of FPGAs in the field. The title of the article is: "Implementation of a Server Architecture for Secure Reconfiguration of Embedded Systems". My contribution is focused on following-up on the server architecture in order to make it suitable for communication with the FPGA.

The elliptic curve processor, discussed in Chapter 2, has been altered to work on binary Edwards curves. This implementation served as a testing-framework in the work of Batina, Hogenboom, Mentens, Moelans, and Vliegen. The results of this work were presented as "Side-channel evaluation of FPGA implementations of binary Edwards curves", at the 2010 International Conference on Electronics, Circuits, and Systems (ICECS). My contribution is the design of the processor together with the implementation of Edwards curves.

1.5 Overview of the thesis

This dissertation consists of six chapters. This first chapter, Chapter 1, contains an introduction that positions the research. Additionally it provides the reader with the most relevant background information with respect to the remainder of the thesis, and with a summary of my contributions. Chapter 2 describes the implemented cryptographic components. These implementations are used in the following three chapters which each describe a specific application. The first application is presented in Chapter 3, which elaborates on the implementation of a secure distributed logging concept. The second application describes a single-chip solution for the secure and remote reconfiguration of deployed FPGAs in Chapter 4. The final application is presented in Chapter 5 and is a prototype implementation of a pay-per-use licensing scheme. The final chapter in this dissertation, Chapter 6, draws conclusions of the carried out research and summarises the different tracks on which future work could elaborate.

1.6 Conclusions

This introductory chapter first gives a motivating example to illustrate how FPGAs can be used to assist in helping hardware follow the evolutions that software has already gone through. Because of the importance of the FPGA in our work, this chapter subsequently introduces the FPGA. Updating already deployed devices requires security. This can be achieved by using cryptographic algorithms and protocols. The relevant aspects of cryptology are introduced in the subsequent section of this chapter. With a focus on applications, the cryptographic overhead has to be as small as possible. This leads to a focus of the implementations in this work on minimal resource usage. Finally, the main contributions of this thesis are summarised and an overview of the dissertation is given.

Table 1.4: Feature overview of recent Xilinx FPGAs

Feature	Virtex-4 & Virtex-5	Spartan-6	Virtex-6 & 7-series
Year of commercial release	2004 & 2006	2009	2010 & 2011
Technology	65 nm	40/45 nm	40/45 nm & 28 nm
Encrypted full bitstream	AES256	AES256[1]	AES256
Encrypted partial bitstream	no	no	yes
Key storage	BBRAM	BBRAM / eFUSE	BBRAM / eFUSE
ICAP readback with encrypted bitstream	yes	yes	yes
ICAP support with encrypted bitstream	yes[2]	yes	yes
Data authentication on bitstream	no	no	HMAC

[1] not in LX, SX and FX12
[2] for Virtex-4: only in LX75(T), SLX100(T), LX150(T)

Chapter 2

Cryptographic components on FPGA

This chapter thoroughly discusses the four implemented cryptographic components that realise basic cryptographic concepts explained in the Section 1.3. These implementations are used in the applications handled in later chapters. The first two components are unkeyed primitives: a cryptographic hash function and a true random number generator. The third component is a public-key primitive while the fourth implementation is a symmetric-key cipher.

These four components are wrapped into a similar interface, which is also presented in this chapter.

2.1 The component wrapper

As mentioned above, every component in this chapter is wrapped into a similar interface. This allows seamless interchangeability between different implementations of a component. Thus, when a specific application requires a component of which the implementation has a slightly different focus, this interchangeability eases the substitution.

The component wrapper provides several inputs. Apart from a general 'reset' and 'clock' signal, an 'enable' signal is available in the interface to start the component. Additionally, a variable number of inputs can be used to parametrise the behaviour of the targeted component.

Next to the inputs, a general 'done' output-signal indicates when a certain component finished its operation.

To provide every cryptographic component with data input and output, a memory interface is provided containing a 32-bit address, a 4-bit write-enable ('we'), a 32-bit data input ('data_in'), and a 32-bit data output ('data_out').

The resulting wrapper is shown in Figure 2.1, where the dashed rectangle is a place-holder for every cryptographic component. Every input and output, discussed above, is shown in this figure. A right-pointing triangle represents an input and a left-pointing triangle represents an output.

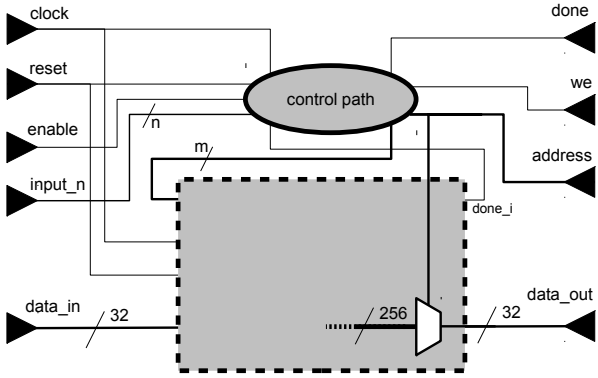

Figure 2.1: Architecture of the component wrapper

Every implementation discussed below fits in this wrapper, with exception of the elliptic curve processor. The latter has the unified interface built-in. With this common interconnect, every cryptographic component can easily be replaced by another implementation. This may be desirable when better or more suitable algorithms or implementations are required.

2.2 Hash / HMAC

A hash function naturally makes a good candidate for verifying data integrity. If the data needs to be authenticated as well, a symmetric-key scenario is appropriate. A cryptographic component that offers the former, with an optional authentication on the data, incorporates both an unkeyed primitive and a symmetric-key primitive into one component. Combining both primitives saves reconfigurable resources.

2.2.1 Secure Hashing Algorithm

There exist many algorithms to calculate a hash on a certain input. The first hash function, standardised in 1993 by the American National Institute of Standards and Technology (NIST), was the Secure Hashing Algorithm (SHA-0). Because it contained an error in the specification that led to significant weaknesses, it was replaced by the very similar SHA-1 in 1995. This algorithm produced a hash value of 160 bits. In 2001, NIST has standardised a set of six cryptographic hash functions under SHA-2 which all produce a hash of length: 224, 256, 384 or 512 [59]. In 2007, NIST announced a competition in search of new hashing algorithms to find SHA-3, which was won by the Keccak algorithm [4] in October 2012.

As a master thesis, Ramakers and Narinx implemented four candidates of the latest hash competition: LANE, Luffa, Hamsi, and ECHO [53]. As a reference they have implemented a SHA-2 function: SHA-256. The first implemented cryptographic component in this dissertation starts from this SHA-256 implementation.

The message is to be segmented to 512-bit blocks. The last block has to be padded as described in the standard [59]. The first block is processed upon receiving a 'start' signal, while other subsequent blocks are started with the continue signal. When a block is parsed, the 'done' comes up to indicate the 256-bit hash is present at the output.

2.2.2 HMAC

In [56], the keyed-hash message authentication code (HMAC) is standardised of which the algorithm was initially published in [40]. In the first pass, the key is exored with a constant (IPAD) and precedes the message. On this, a hash is calculated. In the second pass, the key is exored with another constant (OPAD) and precedes the hash resulting from the first pass. On this new message, a hash is again calculated. The result of the second pass is the HMAC on the original message. This approach is visualised in Figure 2.2.

2.2.3 Implementation of SHA-256/HMAC

The implementation of SHA-256 is straightforward as the standard prescribes. It takes a correctly padded input block of 512 bits and calculates or updates the hash value. When the calculations are done, the 'done' signal comes up and the hash is available at the output. Figure 2.3 shows the top level architecture

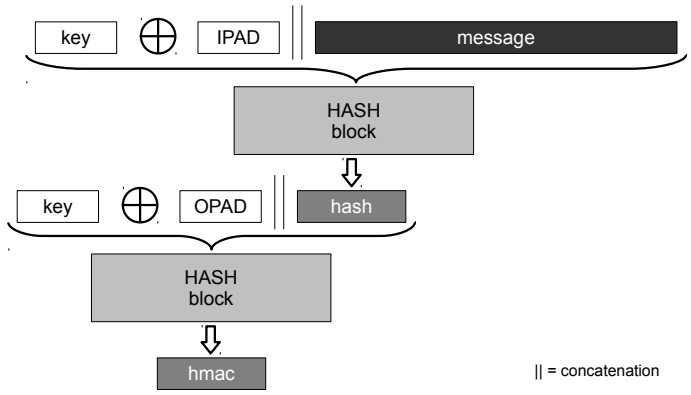

Figure 2.2: Visualisation of the HMAC algorithm

of the SHA-256-implementation from [53]. The dashed rectangle fits the general component wrapper, shown in Figure 2.1.

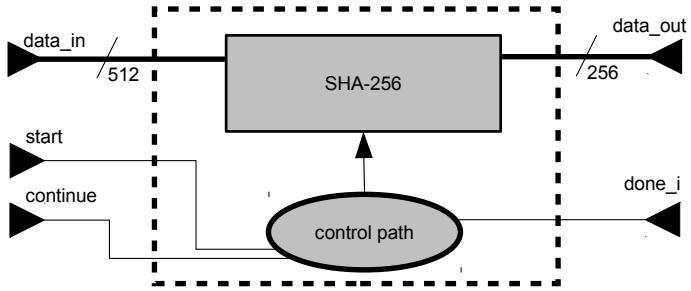

Figure 2.3: Architecture of SHA-256

The architecture of the HMAC implementation is shown in Figure 2.4. The white rectangle in the middle holds the implementation shown in Figure 2.3. The additional components are a 512-bit shift register, two 512-bit exor gates, a single 512-bit multiplexer and a single 256-to-32-bit demultiplexer.

It is pointed out that during the calculation of the HMAC, the output of the first pass has to be padded again and routed to the input to serve as input for the second pass. This step is not implemented in hardware and has to be done by the controlling component.

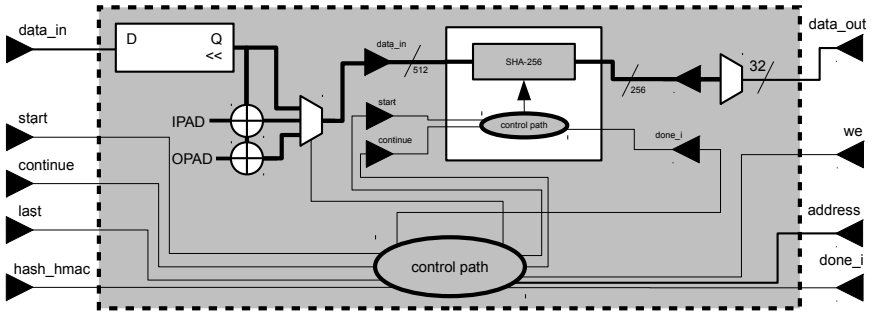

Figure 2.4: Architecture of SHA-256/HMAC

The control path is implemented as a finite state machine (FSM)[4] of which the different states and their succession are shown in Figure A.2 in Appendix A.2.

2.2.4 Wrapper fitting of the SHA-256/HMAC

The dashed rectangle in Figure 2.4, is inserted in the dashed rectangle of the component wrapper, shown in Figure 2.1. Table 2.1 summarises the signal mappings.

Table 2.1: The SHA-256/HMAC signal mapping in the component wrapper

component pin name	wrapper pin name	direction	description
reset	reset	I	active low reset
clock	clock	I	main clock signal
start	enable	I	new hash/HMAC calculation
continue	input_0	I	continue hash/HMAC calculation
last	input_1	I	last hash/HMAC calculation
hash_hmac	input_2	I	indicate hash or HMAC
data_in	data_in	I	32-bit input block
data_out	data_out	O	32-bit output block
we	we	O	4-bit write enable
address	address	O	32-bit write enable
done_i	done	O	component done signal

[4] A finite state machine is a control structure that stores the state of a sequential process at a given time,and can operate on inputs to change the status and/or cause an action or output to take place for any given change.

2.2.5 Implementation results

The results of the implementation of the SHA-256/HMAC component are shown in Table 2.13 on page 52.

2.3 True Random Number Generator

The implementation of the true random number generator (TRNG) uses freely running oscillator rings as physical random source. If a ring of an odd number of inverters is implemented, the output starts switching between a logical '1' and '0'. The speed with which the output switches, is implementation dependent. This patented design was originally proposed by Schulz [67].

In [70], Sunar et al. build on this design to achieve an FPGA-based TRNG. They implemented multiple rings which oscillate at a slightly different frequency. Sunar et al. use this difference as the noise source for the random number generator.

Wold and Tan improved the design of Sunar et al. in [77] by adding a flip-flop after each oscillating ring. By doing so, the authors of [77] enhanced the TRNG with better randomness characteristics and they have no need for post-processing while their implementation still passes statistical tests. With these improvements the design of Wold and Tan is more than a factor 10 smaller in FPGA resources than the work of Sunar et al. Because all implemented cryptographic components focus on area, the TRNG is implemented according to the design in [77].

2.3.1 Implementation of TRNG

Figure 2.5 shows the architecture of the random number generator. The internal grey-coloured boxes indicate possible repetitions which alter the size of the oscillating rings, n_inv, and the number of oscillating rings, n_rings. The large white-coloured box, which indicates the design of [77], produces a single random bit on every clock cycle, which is shifted into a 256-bit register. Upon the request for a random number, this register is sampled and presented at the output. It is pointed out that two requests for a random number need to be separated 256 clock cycles at least. This is to be managed at the controlling level.

The implementation of the TRNG uses 'generics' to determine n_inv and n_rings. The default values of these 'generics' are $n\_inv = 3$ and

$n\_rings = 50$. The reason for choosing these defaults is a trade-off between occupied resources of the implementation and the bias in the generated random bits. Theoretically this bias should be 0.5 [77]. Using the lowest number of inverters in a ring, as reported by Wold and Tan, reduces the occupied resources. According to [77], the bias in 3 inverter oscillating rings approximates the desirable 0.5 best when implementing 50 oscillating rings.

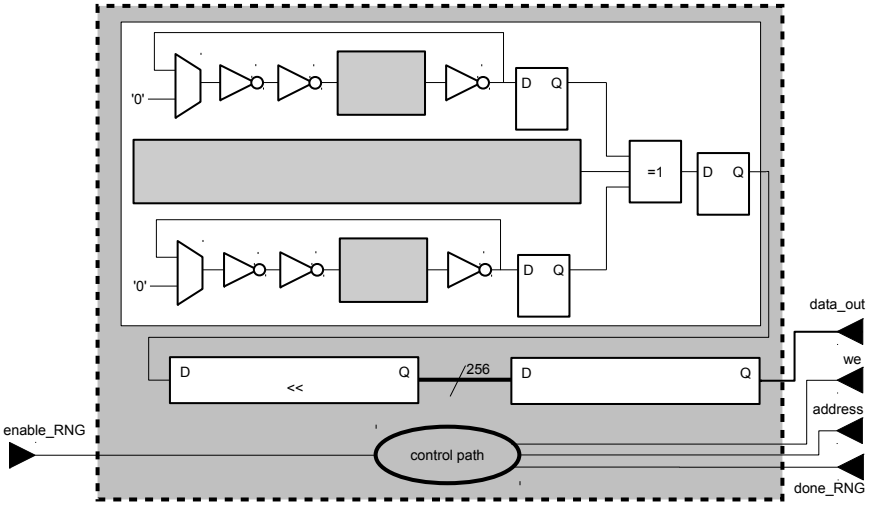

Figure 2.5: Architecture of the Random Number Generator

2.3.2 Wrapper fitting of the TRNG

The outer, dashed rectangle in Figure 2.5, is inserted in the dashed rectangle of the component wrapper, shown in Figure 2.1. Table 2.2 summarises the signal mappings.

2.3.3 Implementation results

The results of the implementation of the TRNG component, with 50 oscillating rings built with 3 inverters, are shown in Table 2.13 on page 52. The reported timing is for a single random bit.

The NIST statistical test suite has been run over approximately 1500 randomly generated numbers. All the statistical tests on these random numbers passed

Table 2.2: The TRNG signal mapping in the component wrapper

component pin name	wrapper pin name	direction	description
reset	reset	I	active low reset
clock	clock	I	main clock signal
enable_RNG	enable	I	random number request
data_out	data_out	O	32-bit output block
we	we	O	4-bit write enable
address	address	O	32-bit write enable
done_RNG	done	O	component done signal

except the non overlapping template test. This might indicate that a small post-processing step is required.

2.4 Elliptic Curve Processor

The third cryptographic component is an elliptic curve processor (ECP) which calculates the modular operations over a prime field $GF(p)$. Although the ECP architecture is scalable to every prime field curve, in the scope of this dissertation it is only implemented for the $P-256$ curve [60]. According to [21] this field size is sufficient to provide long-term protection.

2.4.1 Implementation of the ECP

The implementation of the ECP follows the Harvard architecture, shown in Figure 2.6, which implies having physically separated instruction and data buses, in contrast to the von Neumann architecture. Next to these two memories there is a control unit, an IO unit and an ALU.

The programs that run on the ECP are stored in the instruction memory as hardware instructions, generally referred to as 'microcode'. A small parser script is implemented so the programs on the ECP can be written in machine code or assembly and translated into microcode.

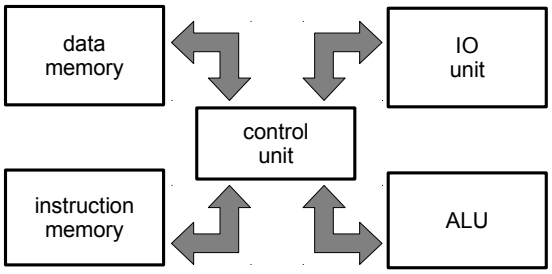

Figure 2.6: Architecture of a Harvard processor

Data and instruction memory

The data memory is built out of nine serial 32-bit BRAMs to provide a maximum data width of 288 bits. All these BRAMs are used as a dual ported RAM.

The instruction memory is a single BRAM configured as a ROM, which contains the programs. The program counter provides the BRAM with an address and the instruction memory returns the microcode instruction which is stored in the instruction register. These instructions are 32 bits wide and represent fourteen operations which are shown in Table 2.3, together with their operation code (opcode), required number of arguments, a mnemonic, and duration to execute. A microcode instruction has 3 reserved bits for future expansion, a 5-bit opcode and 24 bits for operands.

Control unit

The control unit handles the instruction from the instruction memory according to the classical fetch-decode-execute-increment approach. The program counter indicates which instruction to fetch from the instruction memory and to store it in the instruction register. The instruction gets decoded and directs the operation of the necessary components through a finite state machine of which the states and their successions are shown in Appendix A.1. The duration of the execute step depends on which operation is requested. After this execution the program counter is incremented with exception of the execution of a jump or a restore of the program counter. Note that because of this implementation approach, even a 'NOP' requires 5 clock cycles.

The control path of the ECP is an FSM of which the states and their successions are shown in Appendix A.

Table 2.3: Instructions handled by the ECP

assembly mnemonic	opcode	# arg	description	duration [clock cycles]
storePC	00001	2	stores the program counter	5
jmp	00010	1	unconditional jump	4
cjmp	00011	2	conditional jump	5
	0 jump if key(s_data-2 = $argv(2))			
	1 jump if counter $\neq 0$			
	2 jump if counter(8) = 1			
shift	00100	0	shift the scalar	5
exec	00101	5	execute arithmetic operation	[1]
restorePC	00110	1	restore program counter	4
cp2MEM	00111	2	copy data to local memory	14
cp2PT	01000	2	copy data to PT memory	15
load	01010	0	load scalar	6
loadN	01011	0	switch to $GF(p)$	7
loadO	01100	0	switch to $GF(order)$	6
NOP	11110	0	no operation	5
END	11111	0	end of program	4

[1]: depending on the operation

IO unit

To communicate with other components, the ECP has the common interface as explained in the beginning of this chapter. The ECP has two custom instructions to read and write to this external memory interface: cp2MEM and cp2PT, respectively. These instructions read or write a complete 256-bit word to the data memory in units of 32 bits.

ALU

Although no dedicated logical components are implemented, the arithmetic-and-logic-unit (ALU) contains two arithmetic components: a modular adder/subtractor (MAS) and a modular Montgomery multiplier (MM).

The architecture of the modular adder/subtractor is shown in Figure 2.7. The component simply adds 'X' and 'Y' when 'sign' = '0' and stores the result in the data memory. Subsequently the modulus (N) is subtracted from the sum. When the result of the subtraction is positive, the previously stored result

is overwritten; otherwise the write enable signal ('we_o') is kept inactive to feign overwriting. For subtraction, indicated with 'sign' = '0', the procedure is analogous with inverted signs. Negative numbers are represented using the two's complement method. To make a trade-off between area and speed, the MAS works on a part of the operands thus reducing the width of the datapath of the adder. The operands are segmented into words with a width of w bits. The ECP is implemented with an 8-bit and a 32-bit word size w.

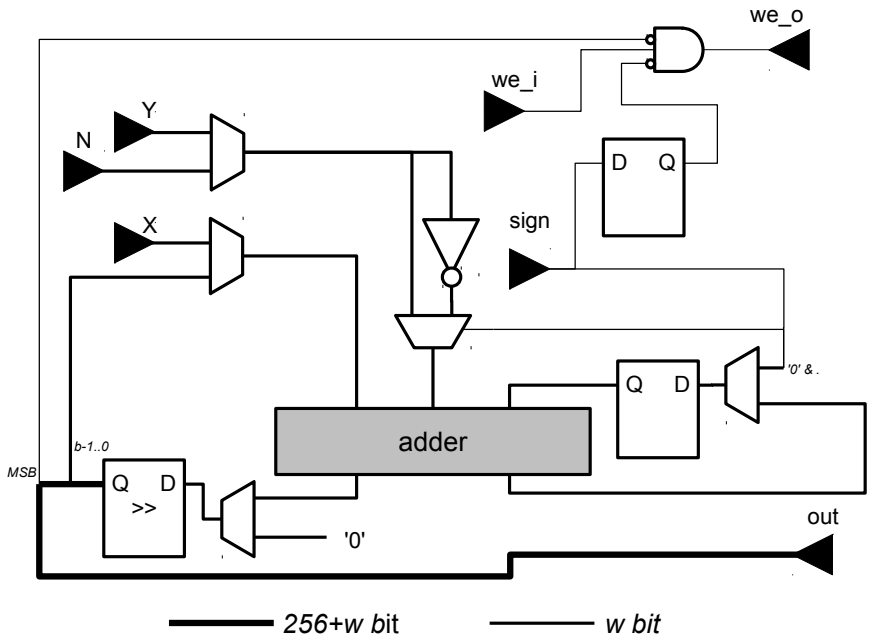

Figure 2.7: Architecture of the modular adder/subtractor

Figure 2.8 shows the architecture of the implementation of the Montgomery multiplier. This architectures multiplies 'X' and 'Y' modulo 'N' and presents the result at 'out'. To overcome the expensive division operation which should be used in modular operations, Montgomery presented a novel technique for calculating a modular multiplication in [49], which is shown in Equation (2.1).

$$\begin{aligned} Z_{Mont} &= MM(X_{Mont}, Y_{Mont}) \\ &= X_{Mont} \cdot Y_{Mont} \cdot R^{-1} \, mod \, p. \end{aligned} \tag{2.1}$$

By applying this technique, the division by the modulus is replaced by a division by a power of two, which comes at a negligible cost in

hardware. To use the Montgomery multiplication (MM), operands have to be transformed into 'Montgomery representation', denoted X_{Mont}, by calculating: $X_{Mont} = x * R \, mod \, p$. It is pointed out that conversions to Montgomery representation and back can be done through the Montgomery multiplication itself: $X_{Mont} = MM(x, R^2)$ and $x = MM(X_{Mont}, 1)$ respectively.

When calculating a modular multiplication using a Montgomery multiplication, a final conditional subtraction is needed [49]. In [76], Walter presents a method that avoids this final subtraction. The cost of this solution is that the width of the intermediate values, in the Montgomery multiplication, increases with one word. The consequence for the size of the data memory, is an increase of one digit size w. Keeping in mind that the field elements are 256 bits, this results in a data memory width of $256 + w$ bits, where w is the size of the datapath of the mathematical components.

In [35], Koç et al. analyse and compare the time and space requirements of five different implementations of a Montgomery multiplication. The ECP has an implementation of the CIOS method because this method requires the least area and the fewest number of additions and multiplications. The implementation of the MM also is word-based, similar to the MAS. For details and formulas to calculate R and N', [35] is referred to.

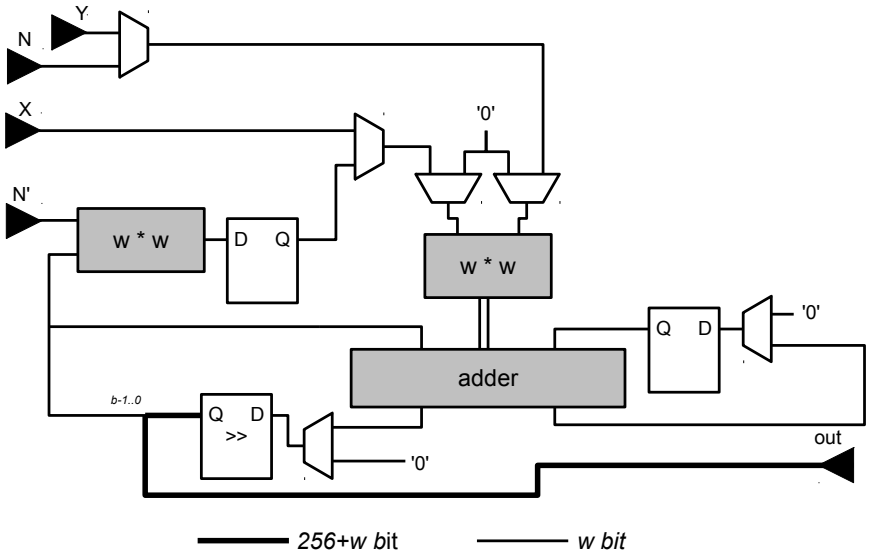

Figure 2.8: Architecture of the modular Montgomery multiplier

Finally, the top level architecture of the implementation of the ECP is shown in Figure 2.9. The implementation has been described using generics to be able

to change the datapath width of the MM and the MAS. As mentiond above, the ECP is implemented with a 8-bit and a 32-bit word size w. To reduce the required resources, both shift registers containing the operands of the MM and MAS are shared. Also the modulus is stored in a shared shift register.

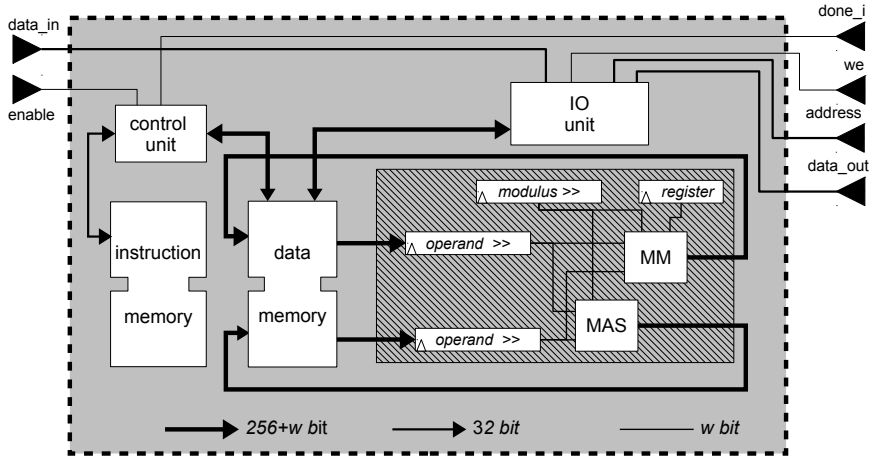

Figure 2.9: Architecture of the elliptic curve processor

2.4.2 Software on the ECP

The ECP can be used to calculate the point additions and point doublings as explained in Section 1.3.3. With these two operations the calculation of a scalar multiplication is feasible.

In certain settings, the scalar is a secret. For example, the generation of a public key A in the elliptic curve public-key pair (A, a) is done through calculating $P \cdot a$. Leaking the scalar during the scalar multiplication would leak the private key, in this case. As explained in Section 1.3.4, SCA-attacks can be a way to find such a secret. Coron described a simple SPA-attack in [12] which is applicable if the scalar multiplication is implemented using the double-and-add algorithm (Algorithm 1). The conditional if-branch is unbalanced in this algorithm. In the case of scalar-bit '1' the extra point addition is executed while it is not in the other case. A visual analysis of the power consumption could reveal the value of the used scalar-bit and thus leak the scalar.

Although not resulting in the value of the scalar, a TA (see Section 1.3.4) could

be used to find the Hamming weight[5] of the scalar which significantly reduces the search space which could allow a brute-force attack. This vulnerability can easily be overcome by using the Montgomery ladder (Algorithm 2 in Section 1.3.3) which balances both branches of the conditional if-branch. Nevertheless, a good focus on the assembly code is required. Table 2.4 gives a code snippet of the assembly code that runs on the ECP. Although line 152 is functionally useless, it compensates an unbalance in the assembly instructions for the execution of the Montgomery ladder. Moreover, in the case of scalar bit value of '1', the code executes a 'STOREPC' and two 'JMP' instructions. Line 152 has to be added to guarantee that the same instructions are executed in the case of a scalar bit value of '0'.

Table 2.4: ECP assembly code snippet for the scalar multiplication

144	STOREPC	0	146		
145	JMP	24			$S \leftarrow 2 \cdots Q$
146	CJMP	150	0	0	jump if scalar-bit '0'
147	STOREPC	0	149		
148	JMP	51			$Q \leftarrow S + Q$,$S \leftarrow 2S$
149	JMP	153			
150	STOREPC	0	152		
151	JMP	87			$S \leftarrow S + Q$,$Q \leftarrow 2Q$
152	JMP	153			
153	SHIFT				
154	CJMP	146	1		jump if bits remaining
155	RESTOREPC	1			

In [31], Izu et al. present improved methods for implementing a scalar multiplication which are resistant against SCA attacks. For the 'Montgomery-type method' they increased the efficiency of the scalar multiplication by encapsulating both formulas of the ECPA and ECPD into one formula. Moreover, they use a projective coordinate system from which they only use the x-coordinate and z-coordinate. This method performs an ECPA and ECPD in 13 modular multiplications, 4 squarings, and 18 additions. To recover the y-coordinate, possibly after multiple point operations, an additional 11 modular multiplications, 2 squarings, and 7 additions are required. As a reference, a comparison with the work of Cohen et al. [11] is shown in Table 2.5.

[5]The Hamming weight is defined as the number of non-zero symbols in a string. For a binary value this comes down to the number of '1' bits.

Table 2.5: Comparison of operations between [11] and [31]

operation	Cohen et al. [11] ECPA	Cohen et al. [11] ECPD	Izu et al. [31] ECADDDBL	Izu et al. [31] Y coordinate
modular multiplication	26	11	17	13
addition	18	9	18	7

Because both Cohen et al. and Izu et al. use a projective coordinate system, an additional bi-directional conversion is required. All required conversions thus are:

- conversion to Montgomery representation: $P(x, y) \rightarrow P(X_M, Y_M)$;
- conversion to projective coordinates: $P(X_M, Y_M) \rightarrow Q(qx, qy, qz)$ where $qx = X_M * qz$ and $qy = Y_M * qz$ with $qz = 1 * R$ which is the Montgomery representation of the value '1';
- conversion from projective coordinates: $P(X_M, Y_M) \leftarrow Q(qx, qy, qz)$ where $X_M = \frac{qx}{qz}$ and $Y_M = \frac{qy}{qz} * qz$;
- conversion from Montgomery representation: $P(x, y) \leftarrow P(X_M, Y_M)$.

In the conversion from projective to affine coordinates, a division is used. Because an implementation for a multiplication is present, this division is calculated by multiplying with the inverse of the denominator. The calculation of this inverse is done by using Fermat's little theorem[6], which substitutes the calculation of the inverse with an involution.

To be able to run multiple programs on the ECP, the instruction memory always starts by making a copy of a value from the pass-through memory to the local memory. This copied value contains an address to which the processor restores its program counter to in the second operation. After this boot-loader mechanism, a reasonable amount of instruction memory is dedicated to general functions which can be used by all programs. Finally, the remainder of the memory is available for main programs. Three programs are implemented: the calculation of an ECPM, the generation of a digital signature using the ECDSA, and the verification of such a signature. This segmentation of the instruction memory space is visualised in Figure 2.10.

[6]Fermat's little theorem: If p is a prime number and a is an integer, then $a^p \equiv a \bmod p$. When $a \nmid p$ then $a^{p-1} \equiv 1 \bmod p$ so $a^{p-2} \equiv a^{-1} \bmod p$.

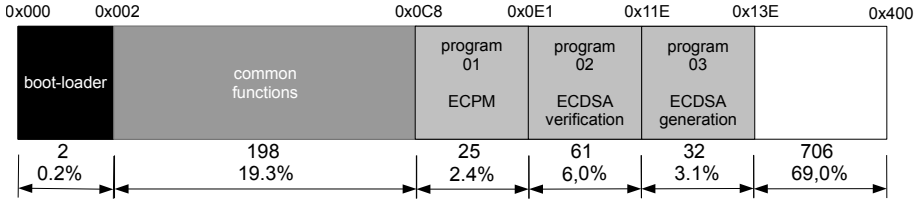

Figure 2.10: Segmentation of the instruction memory

2.4.3 Wrapper fitting of the ECP

The dashed rectangle in Figure 2.9, is inserted in the dashed rectangle of the component wrapper, shown in Figure 2.1. Table 2.6 summarises the signal mappings.

Table 2.6: The ECP signal mapping in the component wrapper

component pin name	wrapper pin name	direction	description
reset	reset	I	active low reset
clock	clock	I	main clock signal
enable	enable	I	start ECP
data_in	data_in	I	32-bit input block
data_out	data_out	O	32-bit output block
we	we	O	4-bit write enable
address	address	O	32-bit write enable
done_i	done	O	component done signal

2.4.4 Implementation results

The discussion of the results of the implementation of the ECP is twofold. First a comparison with other published work is made. Hereafter, a more numerical discussion together with possible improvements is given.

Related work

Most FPGA implementations of elliptic curve processors over prime fields are optimised for speed, while the proposed implementation targets compactness, for reasons explained in Section 1.4.

The design of Örs et al. is based on a systolic array architecture for the modular multiplier [61], while Sakiyama et al. use a course-grained matrix of modular units to perform the modular multiplication [66]. These implementations do not use any dedicated mathematical hard IP blocks, which make them suitable for any FPGA or ASIC. The implementation of McIvor et al. [44] employs the multiplier blocks on the FPGA.

One year after the publication [73] of the implementation made in this work, Varchola et al. presented a solution to obtain a faster and more compact solution [72]. They obtained this by: building an implementation that is restricted to use primes in a special standardised form, by using a better model for the internal adder, and by achieving an implementation at double the clock frequency.

Table 2.7: Implementation comparison with literature

reference	[44]	[66]	[61]	[72]	ECP
# slices	15 755	27 597	*5 614*	1 158	2 085
# MULTs	256	0	0	4	7
# BRAM	0	0	0	3	9
max. freq. (MHz)	39.5	40	*91.31*	210	68.17
ECPM latency (ms)	3.84	17.7	*42.52*	4.52	15.76
FPGA	1	2	3	1	1
year of publication	2004	2007	2002	2011	2010

italic: estimates
FPGA 1: Xilinx Virtex II Pro
FPGA 2: Xilinx Spartan-3 5000
FPGA 3: Xilinx Virtex-1000E

Table 2.7 shows a comparison to related work. The reported numbers for the ECP are those used in its publication, which apply to a Xilinx Virtex-II Pro. This device is no longer supported in the current design tools, but it provides the most honest comparison. The only difference between the implementations on both devices (Xilinx Virtex II Pro and Xilinx Virtex-5) is software handling the Montgomery ladder. The older implementation uses the Montgomery ladder with separate functions for ECPA and ECPD, as reported by Cohen [11], while the other implementation uses the unified formula, as reported by Izu et al. [31].

The three oldest implementations report resources which do not include a communication unit, whereas the two most recent implementations do. The work of Örs et al. also provides support for 2048-bit RSA calculations. Although not implemented here, a small update on the software of the ECP and the

operand sizes provides this implementation with the same feature. This is not possible with the more specialised work of Varchola et al., which outperforms every other implementation with a smaller overhead. Both results of Varchola and the presented ECP, target an implementation with a 32-bit datapath ($w = 32$).

Numerical results

Table 2.8 summarises the clock cycle counts which are required to calculate a certain operation. Because of the processor-architecture there is an overhead of 4 cycles for the fetch, decode, execute and write-back steps (OH_{proc}). The 'execute'-step is used for both the calculation of a MM and a MAS. Because these blocks have a different execution time, an enable-done approach is used. Moreover, when the width of the datapath of a certain block changes, these execution times also vary.

To find the actual duration of an ECPM, all clock cycles have to be added, including those which are processor overhead. This sum has to be multiplied with the period of the actual active clock. Table 2.9 shows the actual and minimal duration of an ECPM, for different datapath sizes.

The FPGA resource requirements for an implementation on a Xilinx Virtex-5 (XC5VFX70T) are summarised in Table 2.13 on page 52. These reported resources reflect an implementation with unified formulas for an ECPA and ECPD.

Possible optimisations

Because of time limitations, several possible optimisations are left unimplemented. Three important ones are explained here.

The FSM which directs the operations of the ECP, can be optimised. The 'dummy'-state can possibly be eliminated, saving a single clock period on every operation (with exception of restorePC and jump instructions). This would reduce the overhead due to the processor architecture.

The software which runs on the ECP can be optimised by in-lining certain functions. Every function call requires three overhead instructions: a store of the program counter in the data memory, a jump to the function in the instruction memory, and a restore of the program counter. On the most repeated functions, this could result in significant gains in duration. To obtain this, the instruction memory will grow due to the in-lining, but from Figure 2.10 can be seen there

Table 2.8: Clock cycle count of ECP operations

operation	$s = (256 + w)/w$
CC_{MAS}	$2 \times (s + 2) + OH_{exec} + OH_{proc}$
CC_{MM}	$[2 \times (s + 1) + 1] \times s + 2 + OH_{exec} + OH_{proc}$
$CC_{A \to PM}$ [1]	$6 \times CC_{MM} + CC_{MAS} + CC_{restorepc}$
$CC_{PM \to A}$ [2]	$4 \times CC_{MM} + CC_{restorepc}$
$CC_{Y\ recovery}$	$13 \times CC_{MM} + 7 \times CC_{MAS} + CC_{restorepc}$
CC_{ECPD}	$13 \times CC_{MM} + 13 \times CC_{MAS} + CC_{restorepc}$
$CC_{ECPA\ \&\ ECPD}$	$17 \times CC_{MM} + 18 \times CC_{MAS} + CC_{restorepc}$
CC_{MI}	$2 \times CC_{MAS} + 256 \times (2 \times CC_{MM} + CC_{shift})$ $+256 \times (2 \times CC_{cjmp} + CC_{shift})$ $+CC_{load} + CC_{restorepc}$
$CC_{MontgomeryLadder}$	$CC_{ECPD} + 255 \times (CC_{ECPA\ \&\ ECPD} + CC_{shift})$ $+255 \times (2 \times CC_{cjmp} + 2 \times CC_{jmp} + CC_{storepc})$ $+(CC_{jmp} + CC_{storepc}) + CC_{restorepc}$
CC_{ECPM}	$CC_{A \to PM} + CC_{MontgomeryLadder}$ $+CC_{Y\ recovery} + CC_{MI} + CC_{PM \to A}$ $+5 \times (CC_{storepc} + CC_{jmp})$ $+CC_{load} + CC_{loadN}$

OH_{proc}: Overhead due to processor = 4 cycles
OH_{exec}: Overhead due to exec command = 2 cycles
[1]: Conversion from affine coordinates to Jacaobian & Montgomery
[2]: Conversion from Jacaobian & Montgomery to affine coordinates
$CC_{restorepc}$, CC_{shift}, CC_{jmp}, and CC_{loadN}: according to Table 2.3

are a little over 700 memory locations still unused. Adding another BRAM to the instruction memory, can double the number of available instructions, if this might be required.

A better implementation for the two implemented ripple-carry adders in the MM and MAS, could lower the area-cost and improve the maximum clock-frequency. The current critical path runs through a ripple-carry adder.

Table 2.9: Actual duration of a single ECPM

datapath width w [bit]	8	16	32
CC_{ECPM}	11 519 044	3 324 244	1 101 916
actual applied clock period [ns]	10.00		20.00
actual ECPM duration [ms]	115.20		22.04
minimum clock period [ns]	8.35		14.85
ECPM latency [ms]	96.19		16.37

2.5 Symmetric-key cipher

As mentioned in the cryptologic background (Section 1.3.2), there exist two types of symmetric-key ciphers: stream ciphers and block ciphers. Table 2.10 shows a small comparison on the differences in both ciphers.

Table 2.10: Comparison between a stream and block cipher

feature	stream cipher	block cipher
scope of encryption	character	block of characters
buffering requirements	low	**fair**
error propagation	low to none	partial to successive[1]
padding	not required	required
data authentication	none	possible
freely available algorithms	few[2]	**many**[2]
reusable	no	yes
	[1]: all blocks after erroneous block [2]: according to Menezes, Vanstone, and Van Oorschot [45]	

Mainly because the FPGA is not too constrained in resources and the number of freely available algorithms, an implementation of a block cipher is made.

Naively using a symmetric block cipher could leak information on what is being encrypted. This can be prevented by using a certain mode of operation. In [58], NIST has published five modes to prevent information from leaking. The first mode should be avoided because it could still leak patterns in the plain text.

Hence, the four possible modes are:

- the cipher block chaining mode (CBC),
- the cipher feedback mode (CFB),
- the output feedback mode (OFB),
- the counter mode (CTR).

Table 2.11 compares these four modes of operation on several implementation features.

Table 2.11: Comparison between four recommended modes of operation

feature	CBC	CFB	OFB	CTR
encryption only	✗	✓	✓	✓
error propagation	✓	✓	✗	✗
encryption parallelisable	✗	✗	✗	✓
decryption parallelisable	✓	✓	✗	✓
random access on decryption	✗	✗	✗	✓
hardware requirements	XOR	XOR	XOR	XOR + COUNTER

Because the CBC mode requires both an encryption and decryption component, this mode does not fit into an area-critical implementation. Although the duration of communication is not assumed critical, squandering time is not acceptable either. Error propagation could render a complete conversation senseless and would require to send the message again, thus wasting time. For this reason CFB is avoided. Having the possibility to parallelise encryption and/or decryption is a valuable feature but comes at an area cost, which is not considered cheap in this work. Random access on decryption is not considered to occur in this thesis so it is not a highly desirable feature. With these two assumptions, the choice on the mode of operation is narrowed down to OFB and CTR. With the focus on area, OFB is the more suitable mode of operation for this work because the CTR mode needs the additional implementation of the counter. It is pointed out that the CBC mode is often used to generate a MAC, referred to as CBC-MAC [54]. Because data authentication can already be provided by the HMAC implementation, the OFB-mode for only providing data confidentiality is sufficient.

The OFB-mode of operation has to be built around the block cipher. The architecture of this wrapper is shown in Figure 2.11. The dashed rectangle fits

the general component wrapper, shown in Figure 2.1. The blue highlighted rectangle is a place-holder for an implementation of a block cipher.

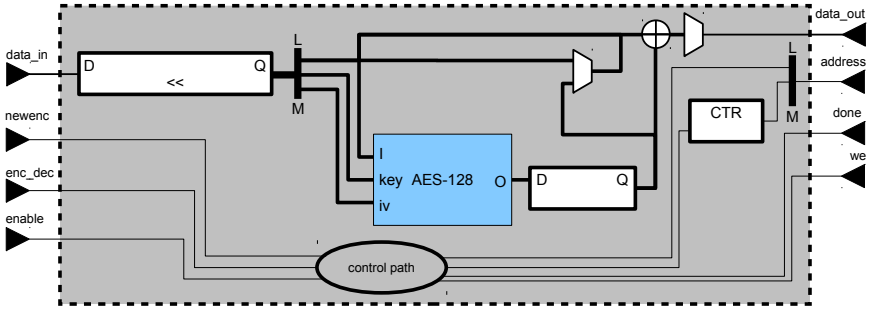

Figure 2.11: Architecture of the OFB-mode of operation wrapper

2.5.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a symmetric-key block cipher which is the current standard defined by NIST [55]. AES is based on the Rijndael cipher [14], which was the winning submission to the AES selection process. Three AES variants exist that all use a block size of 128 bits, but differ in the length of the key. Depending on the length of key, a number of rounds has to be iterated. A single round consists out of four functions: SubBytes, ShiftRows, MixColumns, and AddRoundKey.

Three implementations of AES-128 were made: a small FPGA implementation according to [10], a parallelised implementation of the standard, and a more side-channel robust implementation, based on the work of Moradi et al. [52].

The input is to be segmented in 128-bit blocks. The first block is processed upon receiving an ‘enable’ signal, while the ‘newenc’ signal is kept low. When a block is parsed, the ‘done’ comes up to indicate the 128-bit output is present. Other subsequent blocks are started upon receiving an ‘enable’ signal while the ‘newenc’ signal is kept low.

2.5.2 Implementations of AES-128

The first implementation is based on the work of Gaj et al. [10]. These authors made efforts to have a small implementation on FPGA by moving certain usage of reconfigurable resources to the usage of BRAM. The architecture of the implementation is shown in Figure 2.12 and has a 32-bit datapath.

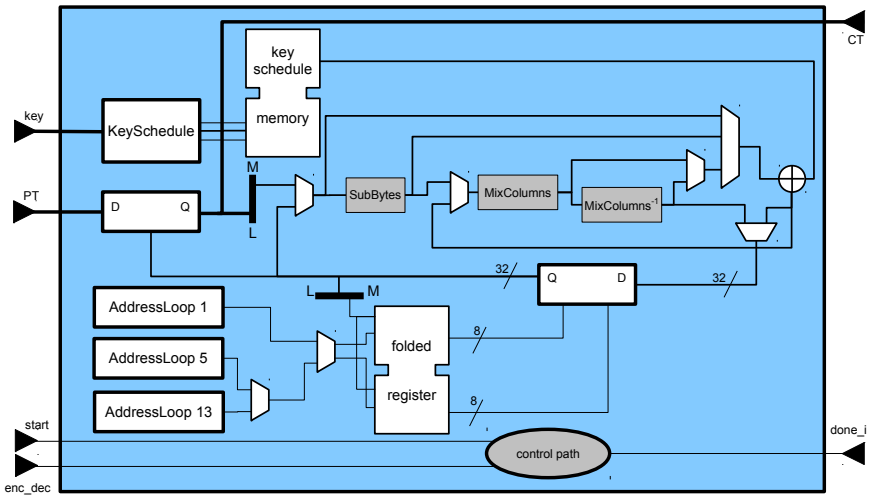

Figure 2.12: Architecture of AES-128 implementation, according to [10]

Subsequently, the second implementation of AES-128 was realised in collaboration with two Master students in the course of their master thesis [71]. The datapath is 128 bits and focused to be able to run at 100 MHz clock frequency. Figure 2.13 shows the implementation architecture of the second AES-128 implementation.

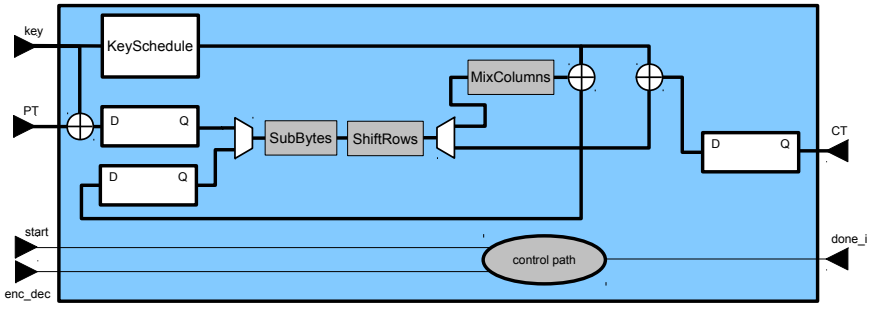

Figure 2.13: Architecture of second AES-128 implementation

Finally, the third implementation of AES-128 is as described in [52]. This implementation is very small and has countermeasures against first order side-channel attacks. The top level architecture is visualised in Figure 2.14.

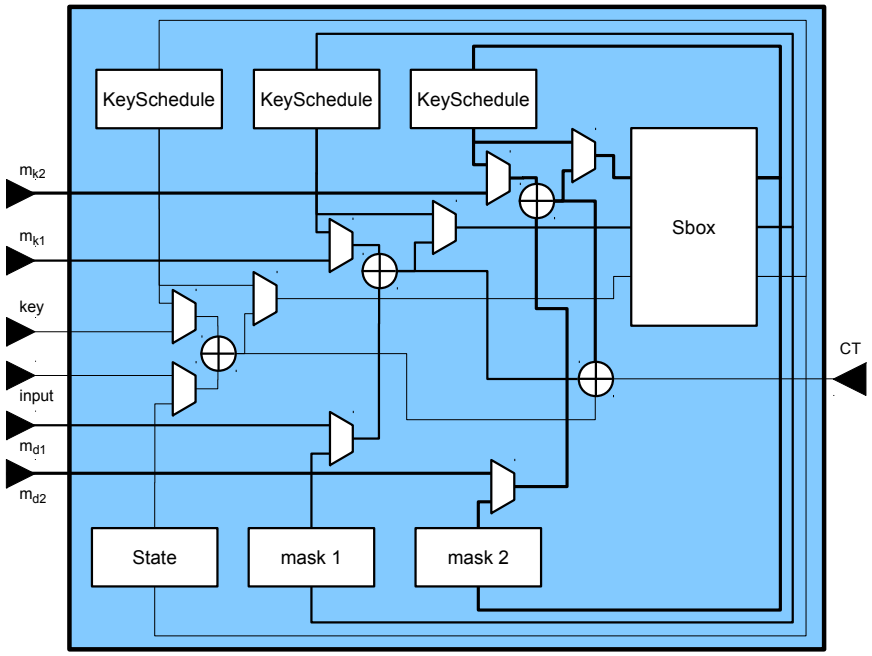

Figure 2.14: Architecture of AES-128 implementation, according to [52]

2.5.3 Wrapper fitting of the AES-128

The dashed rectangle in Figure 2.11, is inserted in the dashed rectangle of the component wrapper, shown in Figure 2.1. Table 2.12 summarises the signal mappings.

2.5.4 Implementation results

All three implementations, wrapped in the OFB wrapper as shown in Figure 2.11, were implemented and successfully tested. The results of these implementations are shown in Table 2.13 on page 52.

It is pointed out that 'impl41' could be further improved. As presented by Hwang et al. in [28], the number of clock cycles for a 128-bit wide implementation of AES can be reduced to 11 cycles. Because of the component wrapper, there are 12 additional clock cycles required for reading the input and 8 additional clock cycles for writing the result. This shows that an implementation that

Table 2.12: The AES-128 signal mapping in the component wrapper

component pin name	wrapper pin name	direction	description
reset	reset	I	active low reset
clock	clock	I	main clock signal
enable	enable	I	start AES-128
newenc	input_0	I	indicate new AES-128 block
enc_dec	input_1	I	indicate encryption or decryption
data_in	data_in	I	32-bit input block
data_out	data_out	O	32-bit output block
we	we	O	4-bit write enable
address	address	O	32-bit write enable
done	done	O	component done signal

only takes 31 clock cycles should be feasible instead of the 63 clock cycles that 'impl41' uses.

2.6 Summary of the implementation results

Table 2.13 gives a summary of the implementations described in this chapter, implemented on a Xilinx Virtex-5 (XC5VFX70T).

2.7 Conclusions

Four cryptographic components are presented in this chapter. Every implementation foresees a similar interface so updating an implementation with a more suitable algorithm or better performance is fairly easy. These four components will be used in the following chapters where applications are discussed, that build upon these implementations.

The elliptic curve processor was published in the proceedings of the 21*st* IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP) [73].

Table 2.13: Summary of the implementation results on XC5VFX70T

	Occupied slices	36K BRAM	DSP 48E	min clk period [ns]	num of clock cycles	latency [ns]
impl10	671	0	0	9.06	94	852
impl20	316	0	0	3.36	9	31
impl30	1 116	10	7	14.85		
impl31	1 016	10	2	8.35		
impl40	358	10	0	50.36	610	30 720
impl41	1 121	4	0	7.09	63	447
impl42	1 029	0	0	4.46	330	1 472
XC5VLX20T	3 120	26	24			
XC5VFX70T	11 200	148	128			
XC5VLX330T	51 840	324	192			

impl10: SHA-256, straightforward as in [59]
impl20: RNG, straightforward as in [77]
impl30: ECP_32_+RCA_*RCA
impl31: ECP_8_+RCA_*RCA
impl40: AES-128, as Gaj et al. [10]
impl41: AES-128, parallel
impl42: AES-128, as Moradi et al. [52]

Chapter 3

Application: Distributed privacy-preserving log trails

This chapter describes the first application of this thesis: a proof-of-concept implementation to strengthen a distributed privacy-preserving log trail scheme. The existing scheme is designed by Pulls and Wouters and is employed as described in [64]. In a first section the scheme is situated and briefly explained. The second section describes our contribution: the implementation details of the hardware strengthening. This strengthening is achieved by moving the memory component, which contains the most sensitive data of the logging scheme, and the cryptographic primitives from the log server to hardware. An additional benefit from moving the cryptographic primitives is to avoid a communication bottleneck. This chapter ends with a discussion of the results and a conclusion.

3.1 Introduction

3.1.1 Situating the application [75]

In the contemporary online world, digital natives are well aware about their privacy. They tend to be better at this than their parents, the digital immigrants, who sometimes unconsciously share sensitive information to all of their 'friends', if not the world. While this attitude protects digital natives from their peers and other ordinary users of social networks, it does not protect them against the entities that offer these online services to them. Even if these services are very

well protected against attackers trying to harvest information about individuals, a problem exists within the entities that operate these services. Companies such as Facebook and Google are hosting a huge amount of valuable data that can be used for profiling, directed marketing, and other purposes.

In today's cloud services, processes are distributed by default. For individuals it is interesting, and even a basic right [1], to see how their data is handled, with whom it is shared, and how this constitutes a process across several entities. This also makes them aware of potential privacy problems. For entities that are handling these data, such openness can be a strong selling point. Privacy-enhanced log services with log trail reconstruction, or 'distributed privacy-preserving log trails' as Pulls and Wouters refer to it [64], can be a solution to enable such feedback in a privacy-friendly way.

The entities and the dataflow of the scheme presented in [64] are shown in Figure 3.1. The data to be logged are sent by the service provider itself to an external log server, that will generate metadata that links log entries in two chains: one for the affected user and one for the service provider. Afterwards, the user can reconstruct his/her chain using the metadata added by the log server. This is based on a public key pair between the user and the service provider and a shared symmetric key between the service provider and the log server. The privacy of the user is protected by cryptographically hiding that a certain set of encrypted log entries refers to him/her. Once generated and stored, log entries in the storage will only reveal information about their payload, affected user, or the service provider that generated the entry to entities in possession of both the correct secret key used to generate the metadata, and the private key to decrypt the log entry content. Additionally, the scheme allows for third party audits.

One rule of the scheme imposes that the log server will irretrievably overwrite the current value of the shared secret key with its hash value, each time a new log entry is added for the affected entity. Because of this, a compromised or curious log server could − in theory − break some security and privacy properties of the system by retaining state information that is supposed to be deleted as part of the scheme. To reduce the minimum necessary level to which the log server has to be trusted, the functionalities that deal with the state of the log server should be moved to a trusted and certified hardware module. A proof-of-concept implementation for such a hardware module is described in this chapter.

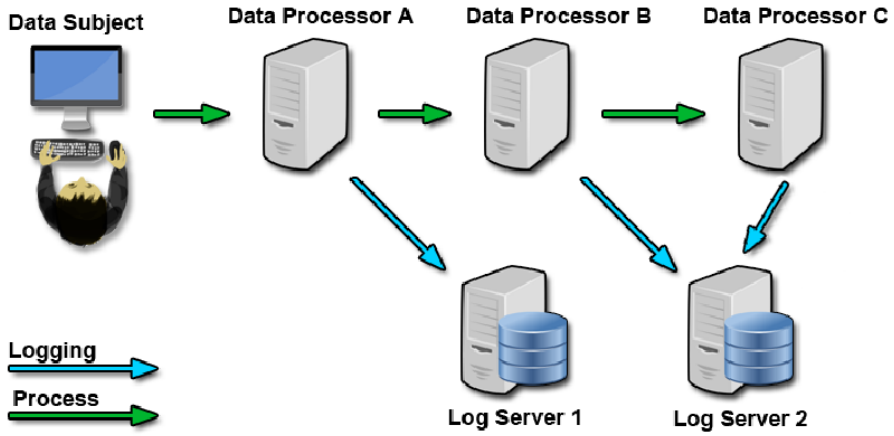

Figure 3.1: The entities and data flow in distributed privacy-preserving log trails

3.1.2 Threat model

It is assumed that an attacker can store every message that is sent between the user, the service provider and the log server, and that he/she also has physical access to the log server. This allows him/her to run custom software on the log server. On top of this, we also assume that the attacker has access to the hardware described in this chapter and that he/she can perform basic power analysis attacks.

3.1.3 A summary of the scheme

To be consistent with the published work [64], the service provider will be referred to as the data processor P_α and the user as the data subject S_α. An essential component in the scheme is the *state component*, shown in Figure 3.2. The state component stores an identifier (ID_{E_α}), a data chain value (DC_{E_α}), an identifier chain value (IC_{E_α}), and an authentication key (AK_{E_α}). Note that E is a placeholder for either P or S. Every time a log server has to handle a new action, the data subject state is updated using the incoming data. These results are used together with the incoming data to update the data processor state analogously. Subsequently, both results are used together with the incoming data to update the log entry. Figure 3.3 shows the state component and its update mechanism for a data subject state.

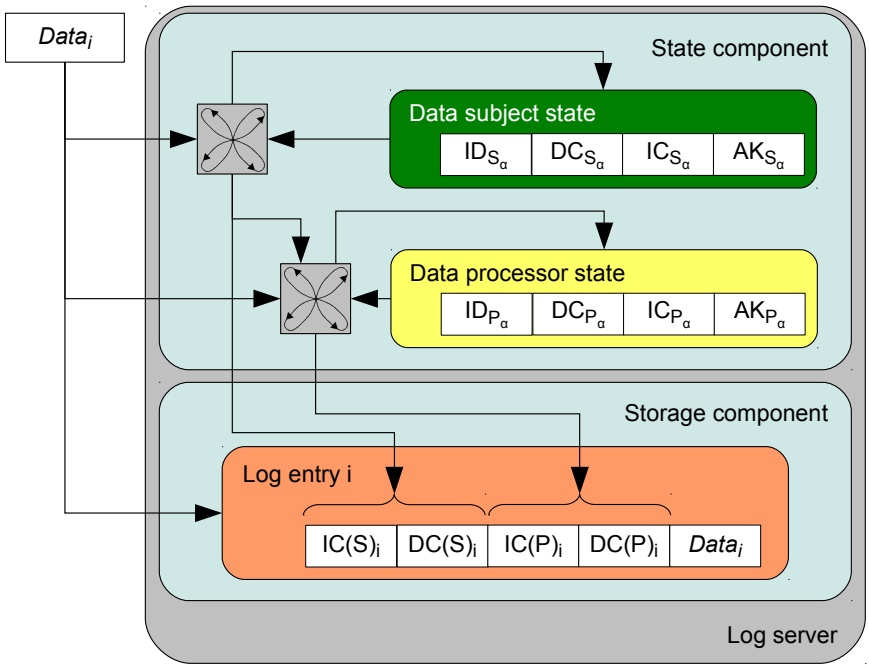

Figure 3.2: The state and storage components in the log server

For every new process of data subject S_1 that starts at a data processor P_1, a new key pair (PuK_{S_1}, PrK_{S_1}) needs to be generated by S_1. The public key, PuK_{S_1}, of the data subject S_1 is transmitted to the data processor and serves as the identifier for the process. Additionally, $PuK_{S_1}$is used for the encryption of the feedback from the log entries. It can happen that a process moves from one data processor P_1 to another P_2. When this occurs, the identifier PuK_{S_1} should be changed as well to ensure unlinkability over data processors and log servers. Pulls and Wouters refer to this as *cascading*. They achieve this by deriving a new public key PuK'_{S_1} from PuK_{S_1} in such a way that it cannot be linked to the original public key, PuK_{S_1}. Moreover, deriving the corresponding private key, PrK'_{S_1}, requires knowledge of the original private key, PrK_{S_1}.

3.2 Implementation

To strengthen the distributed privacy-preserving log trail scheme, the state component is to be moved from the log server to dedicated hardware. Thus, the

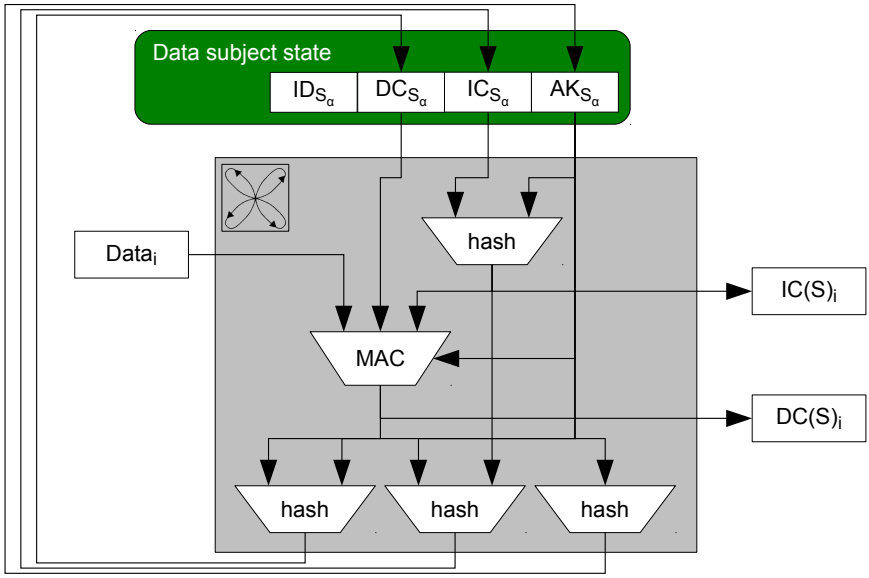

Figure 3.3: The state component for a data subject and its update mechanism

amount of trust that needs to be put in the log server is reduced. Only moving this memory component would not only undermine the strengthening but it would also create a substantial overhead in communication between the log server and the hardware component. To avoid this situation, the cryptographic components, which work on the data in this memory component, are moved to hardware was well. This resulting hardware implementation is referred to as the *trusted state component*, which is used in a system-on-chip. This system-on-chip will be used by the log server. The top level architecture of the trusted state component is shown in Figure 3.4.

A MicroBlaze, which is a 32-bit softcore[7] processor from Xilinx, lays at the centre of the system-on-chip. This central processing unit (CPU) manages the communication with the log server and directs the underlying cryptographic primitives. The MicroBlaze uses a single dual-ported BRAM, which serves as data and instruction memory, each with their own bus. The processor local bus (PLB) is used to interconnect the trusted state component and Ethernet peripherals to the MicroBlaze.

To communicate with the log server, there is a communication channel to the MicroBlaze using the UDP/IP protocol over Ethernet. The reason for choosing

[7] A softcore is an IP core that is implemented on the reconfigurable resources of the FPGA. The alternative is a hardcore IP block which is physically present on the die of the FPGA.

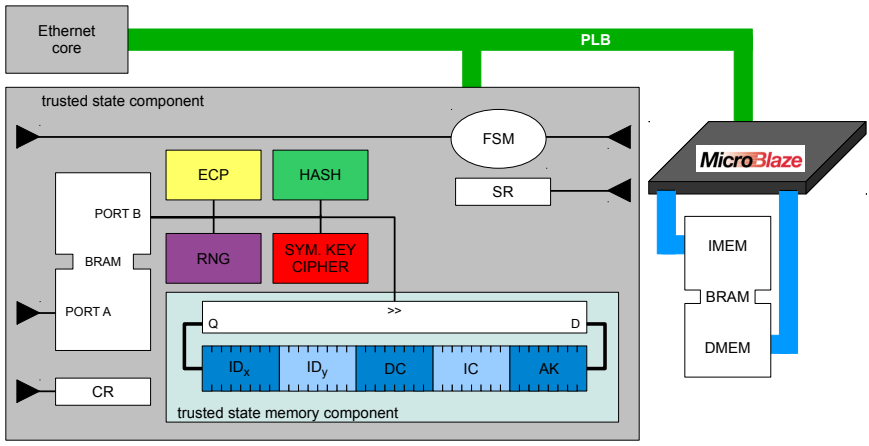

Figure 3.4: The top level architecture of the trusted state component

UDP over TCP is twofold. The stack to handle the UDP protocol is much smaller than the stack of TCP. This is because UDP offers less features than TCP, for example in the TCP protocol every packet has to be acknowledged at reception. Having to manage a smaller stack increases the number of packets the MicroBlaze can handle. The second reason is that the packet header of the UDP protocol is smaller than the header of the TCP protocol (8 bytes and 20 bytes respectively). The application programming interface (API), as described in [64], provides five methods:

- *initialiseProcessorIdentifier* - Initialises a data processor identifier in the log server's state.
- *initialiseSubjectIdentifier* - Initialises a data subject identifier in the log server's state.
- *createEntry* - Creates a log entry, which results in an update of the trusted state.
- *endEntityIdentifier* - Removes the entry for the specified identifier in the state.
- *getStateIC* - Allows the log server to read the content of the trusted state. This feature is required to provide an auditable system.

A detailed look at the functional requirements of the API methods in [64], learns there is a need for six cryptographic primitives: a public-key cipher,

a digital signature component, a hash component, a MAC component, a random number generator, and a symmetric-key cipher. Making an implementation for this scheme comes with a need to make choices for these components. For those components for which a NIST defined standard is available, a choice was made to follow the current standards. Hence, the chosen algorithms are: Digital Signature Algorithm, SHA-256, AES-128, and HMAC based on SHA-256. A choice for the public-key cipher is made to use the Integrated Encryption Scheme [68], the security of which is based on the Diffie-Hellman problem [18]. To be more precise, a variant of the Integrated Encryption Scheme for usage with elliptic curves exists is used and is called ECIES. The requirements for ECIES are a random number generator, a symmetric-key cipher, a key derivation function, a MAC component and an elliptic curve processor.

Except for the key derivation function, an implementation for every component is discussed in Chapter 2. The ECP in the trusted state component has a 32-bit datapath and the AES-128 implementation is the small implementation, based on the work of Gaj et al. [10]. The algorithm chosen as key derivation function is KDF1, which is defined in ISO-18033-2 [30].

Implementations of these chosen algorithms are described in Chapter 2 with exception of the trusted state memory component, that is described in Section 3.2.1.

As is shown in Figure 3.4, these five components are interconnected through a shared BRAM, for which the component wrapper has an interface, and are coordinated by an FSM. The trusted state component is interfaced through a command and status register, that can be written and read over the PLB bus. The executed tasks upon receiving API method calls are visualised in Figure 3.5. Grey coloured rectangles are functions that are executed on the MicroBlaze. The coloured rectangles are cryptographic components, assisting the MicroBlaze in the tasks to which they are attached.

3.2.1 Trusted state memory component

The trusted state memory component is implemented in BRAMs. This implementation is done on a Xilinx Virtex-5 FPGA (XC5VFX70T) which provides BRAMs with a size of 36 Kibit. The maximum available width is 36 bits, which allows a depth of 1024 addresses. This 36-bit word holds four parity bits, which reduces the actual amount of data to be stored to 32 bits per address.

The width of the trusted state memory component is 1280 bits. This is composed by two 256-bit coordinates for the point on the elliptic curve, which serves as

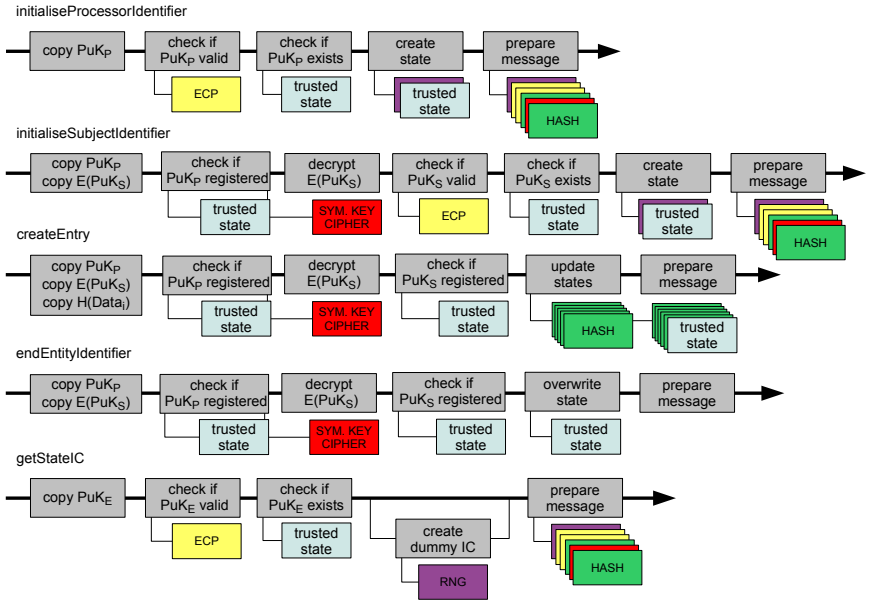

Figure 3.5: Handling of the API methods by the MicroBlaze

a public key, and three times a vector with a width the size of the hash value. Because the actual data width of a BRAM is 32 bits in total, the trusted state memory component needs forty serial coupled BRAMs to be able to store a state on a single address. The trusted state memory component stores and loads data to/from the shared memory, as do the other cryptographic components and it provides storage for 1024 log entries. Since, from a technical point of view, there is no difference between an entry in the trusted state for a data subject or a data processor, the memory space can be segmented to separate both types of entries in the trusted state. The most significant bit of the address thus determines its functions. An additional advantage of segmenting the memory space, is that only a segment of the memory space has to be searched during the lookup of a certain entry. A drawback however is that the ratio processor/subject states is fixed and will have to be carefully predicted. Because this chapter describes a proof-of-concept implementation, this ratio is hard to predict. For this implementation the ratio is assumed to be 1.

3.2.2 Software on the CPU

To initialise the MicroBlaze with custom source files, the toolflow depicted in Figure 3.6 must be followed. The user's source files are compiled into object files. These get linked with a library that contains the drivers of the used (Xilinx) IP cores, and with a linker script. Both the library and the linker script are generated by the Xilinx Software Development Kit, which is part of the Xilinx ISE Design suite. The result of the linker step is an executable that runs on the MicroBlaze.

This executable is to be stored as the initialisation of instruction and data memory in the bitstream. This is done with the 'data2mem' program from the Xilinx ISE Design suite and requires an additional BRAM memory map file (.bmm extension). This bmm file describes how individual block RAMs constitute a contiguous logical memory space and it contains which BRAM in the FPGA is used for the instruction and data memory. The 'data2mem' program results in a bitstream for which the MicroBlaze has an initialised instruction and data memory.

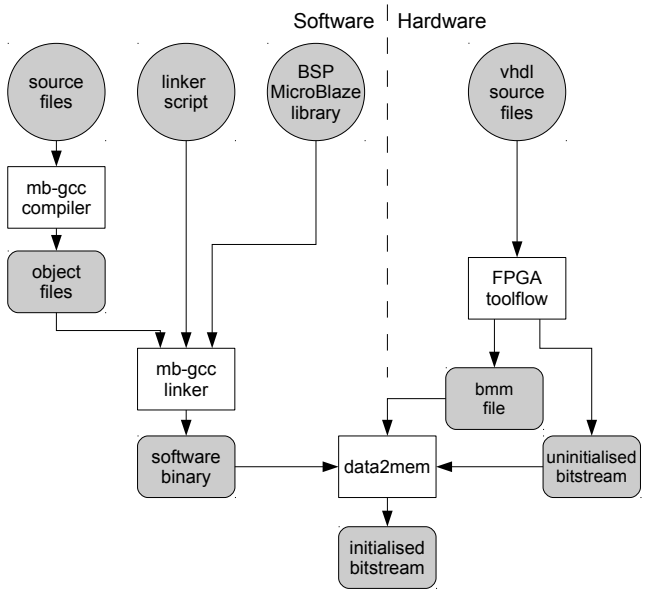

Figure 3.6: Toolflow to initialise the instruction/data memory of the MicroBlaze

As mentioned in the beginning of this section, the MicroBlaze communicates with the log server using an Ethernet connection with the UDP/IP protocol. The software running on the MicroBlaze is written in C and starts by initialising the

Ethernet softcore, which is a Xilinx IP core. Secondly, the constant parameters are initialised, which are mainly used for program offsets, network settings, padding messages for the HASH/HMAC component and the keys used in the scheme. After the initialisations, the MicroBlaze keeps polling the Ethernet component for incoming instructions, following a flowchart as depicted in Figure 3.7.

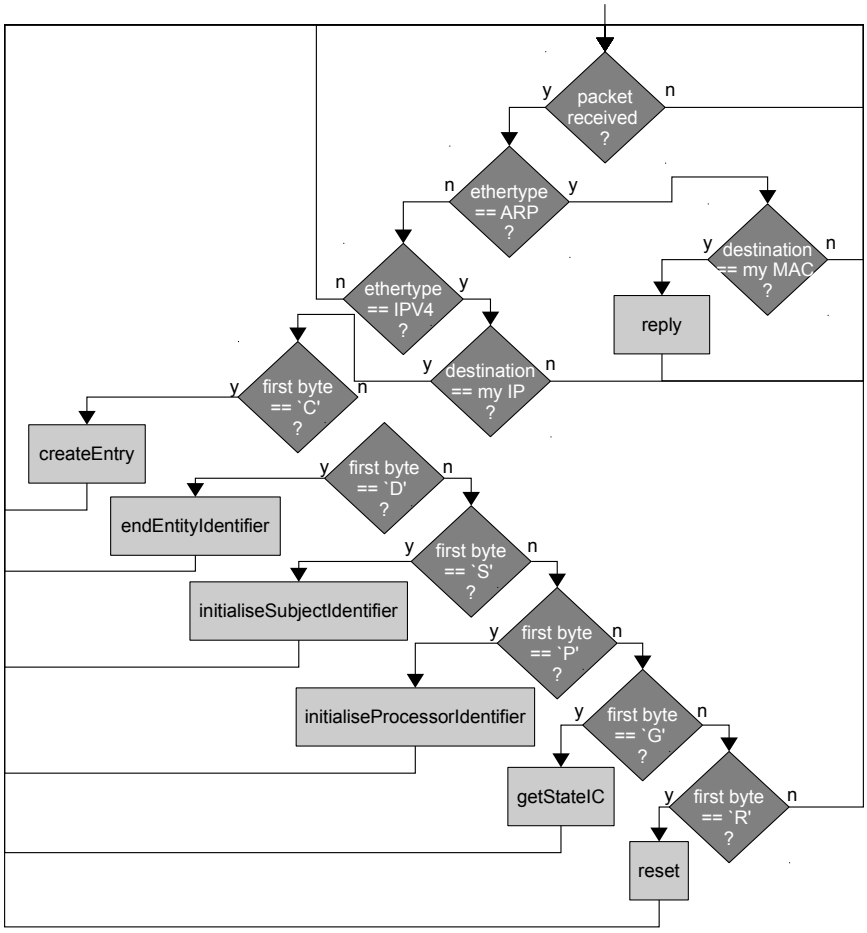

Figure 3.7: Flowchart of the software running on the MicroBlaze

The flowchart first handles requests using the address resolution protocol to find the Media Access Control (MAC) address of the network interface card with a certain IP address. When an IPv4 packet, destined for the MicroBlaze, arrives; the requested API method is looked for in the first payload byte. Depending on

this command, one of the threads in Figure 3.7 is followed. After completing the requested API method, the MicroBlaze starts waiting for a new request again.

3.2.3 Implementation results

Area

The number of used resources for a trusted state component are shown in Table 3.1. The trusted state component is built with 'impl10', 'impl20', 'impl30' and 'impl40' from Table 2.13 at page 52. The complete system uses 4800 slices and 71 BRAMs, while the trusted state component itself takes 2956 slices and 61 BRAMs.

Table 3.1: Occupied resources in a Virtex-5 FX70T

	# slices	# BRAM
trusted state component	2 956	61
complete system	4 800	71
available	11 200	148

As could be expected, the BRAMs are the most critical resource. It is clear from Table 3.1 that a second trusted state component could be implemented in the same FPGA. Even with a second component, there is a little less than half of the number of slices of the FPGA available. This illustrates there is still area available to speed-up the hardware implementation with larger components or to parallelise components. Another option could be to use distributed memory.

Timing

The time it takes to execute one API command can be calculated by counting clock cycles. The system-on-chip works on a 100 MHz clock frequency, which corresponds to a period of 10 ns per clock cycle. The counting of clock cycles starts when the command has arrived at the MicroBlaze and stops after the response message is generated. This way, only the duration of the actual calculations is timed. The additional communication cost in timing has to be added to obtain the entire duration. It is pointed out that the trusted state component operates at 20 MHz to meet the timing constraints of 'impl40' and 'impl30'. This is further elaborated in Section 3.3.2.

Table 3.2: Timing results of the hardware implementation and software reference of the five API methods

	Hardware				Software
	Target at 0		Target at 511		
	small scalar [ms]	big scalar [ms]	small scalar [ms]	big scalar [ms]	[ms]
initialiseProcessorIdentifier	68.47	167.10	79.66	178.29	59
initialiseSubjectIdentifier	68.72	167.35	91.10	189.73	60
createEntry	1.07		23.45		0.062
endEntityIdentifier	9.71		20.90		0.020
getStateIC	0.37		22.75		41
clock frequency	20 MHz				2.2 GHz

Table 3.2 gives multiple results for the duration of a single command on the hardware. The reason for these multiple timings is twofold. The first reason is due to a side effect in the elliptic curve processor. When the ECP performs a scalar point multiplication, the scalar has to be scanned bitwise, starting from the most significant '1', skipping all leading '0' bits. As a result the total execution time of a point multiplication depends on the scalar.

When encoding a message with ECIES, two such scalar point multiplications (with the same scalar) are calculated. This common scalar is a random number. Hence, it cannot be predicted exactly how much time a certain instruction takes. Two extreme cases are when the scalar is 0x1 (small scalar) and when the most significant bit of the scalar is '1' (big scalar).

A second reason for the multiple results for the duration of a single command is the location of a trusted state in the trusted state memory. When a new state (both for data processor and data subject) is to be initialised (or to be searched for) the MicroBlaze has to check the identifiers. To check these identifiers, a linear search in the trusted state memory is performed. This again gives a minimum and maximum delay, depending on the location of the target in the memory space.

Table 3.2 also gives the duration of the different methods with a software-only version of the distributed privacy-preserving logging scheme, which was developed at Karlstad University. The software implementation is written in Java and runs on an AMD Turion X2 Ultra Dual-Core Mobile ZM-82 2.2 GHz processor. From this table it is clear that the hardware implementation slows

down the logging system. The differences in the initialisation of the trusted state are minimal. This is because the hardware executes the elliptic curve operations much faster than the software version. On the other hand, the difference in duration of the 'EndEntityIdentifier' method is considerably larger. Since this method will not be used as often as others, it is not considered as a big issue. The getStateIC method might be the second most (or in some settings even the most) called method. Hence, a faster execution of the getStateIC method on hardware, is an additional gain of the hardware strengthening. The biggest issue, on the other hand, is with the createEntry method.

By comparing the timing results from Table 3.2 with Figure 3.5, it can be concluded that, when only the symmetric cipher, the hash function and the MAC function are called by a method, the hardware performs not as good as software. When the elliptic curve operations are required in a method, the hardware implementation's delay is comparable. The reason for this is that the calculations of ECP are faster on hardware than on software. The loss in time of the symmetric-key cipher, the hash, and the MAC is compensated by the gain in time of the ECP.

Table 3.2 also gives the clock frequency on which the hardware and software are operating. Shifting the focus of the implementations more to speed, can contribute to narrowing the performance gap between the hardware and software versions. Table 3.1 indicates there are resources available to achieve this.

3.3 Results

With the implementation, as described in Section 3.2, we achieve both goals of moving the location of the state memory from the log server to hardware, together with the implementations of the cryptographic coprocessors. The result is the trusted state component.

3.3.1 Additional threats

This section discusses newly added threats and their impact on the overall system. The main additional threats are:

1. Denial of service attacks by interfering with the interconnect.
2. Timing attacks to locate an entity in the memory.

3. Possible loss of the complete internal state because the Block RAMs in the FPGA are volatile.

4. Storage of key material in the bitstream.

Solving the first additional threat is a difficult problem, for which no immediate solution exists. A malicious user, who wants to tamper with the log server by interfering with the interconnect, obviously could interfere with other components in the log server. Therefore it is quite futile to protect against such a threat without physically restricting physical access to the log server and the attached hardware.

The second introduced threat is a TA. Whenever a data processor (or data subject) is registered to a log server, a new trusted state is entered in the trusted state memory component. This means that these entities are stored in a chronological order. A honest-but-curious log server could time how long it takes to lookup an entity in the trusted state to determine for which entity another event is logged. This is a direct threat to the anonymity of the entity in the case of data subject. To solve this threat, a periodical shuffle could be introduced which runs independently from the log server and switches the locations of some values in the trusted state component. By doing so, the honest-but-curious log server can never be sure whether or not a certain entity still is at a certain address in the memory and therefore the timing attack is rendered useless.

The third threat comes from the volatility of the BRAM, which leads to a complete data loss whenever there is a power failure. A solution is to copy the states to an external, non-volatile memory (NVM) e.g. Flash memory. Since it is not acceptable for other people (or their servers) to read these data, they should be encrypted using a public-key cipher for which the trusted party owns the private key. This encryption could be directed by the MicroBlaze, using the cryptographic components of the trusted state. To write the trusted state memory to an external memory, the device still needs a power supply. Different possibilities are at hand. One is to depend on the fact that most servers with important jobs are connected through an uninterruptible power supply. As soon as the log server receives a signal that a power failure occurred, it should send an extra command to the FPGA, indicating not to accept any more changes and to store the states on NVM. Another solution is to provide the required energy through a dedicated component (e.g. a battery or a supercap). As soon as the custom circuit detects a power failure, the energy storage will provide power. Thus, the FPGA can encrypt and write the content of the trusted state memory to NVM.

The storage of the key material in the bitstream is a fourth threat. This manifests itself in the use of ECIES and in the potential backup mechanism. The ECIES algorithm requires the presence of two vectors (S_1 and S_2) which are saved as constants in the instruction memory of the controlling CPU. Eavesdropping the configuration channel could reveal the values of these two vectors which would break the public-key encryption. The key of the potential backup mechanism, would also be exposed to this same threat. A solution for the latter might be obtained from Physical Unclonable Functions (PUFs) [23]. An implementation of a PUF and a KDF can be used to obtain the key for the backup mechanism, thus avoiding the need for storing this key in the bitstream.

3.3.2 Performance bottlenecks and improvements

Although a working proof-of-concept implementation is achieved, the performance can be further improved. This led to a profiling of the implementation to find the bottlenecks. The most important issues are:

1. the presence of a DCM in the trusted state peripheral
2. the unnecessary excessive use of trusted state entity lookups
3. the unnecessary excessive use of memory copies in the shared memory
4. the use of Ethernet

The first bottleneck is due to the presence of a digital clock manager (DCM). This DCM was required to meet the timing constraints of the AES and the ECP implementations, 'impl40' and 'impl30' from Table 2.13 respectively. The available clock signal has a frequency of 100 MHz, which has to be scaled down to 20 MHz to meet all the timing constraints. The clock signal that goes to the trusted state component is hence five times slower than its controlling entity. Additionally, clock cycles are lost for the synchronisation of the command and status registers between both clock domains. If only the ECP and AES implementations were to be moved to a slower clock domain, the area overhead for the synchronisation between the two clock domains would be substantial, due to the synchronisation of all their connections to the shared memory. To prevent this, a choice was made to move the complete trusted state component to the slower clock domain. This way only the status and command register need to be synchronised. This bottleneck can be overcome by using faster implementations for the ECP and AES. Using 'impl31' and 'impl41' in the trusted state component would allow a clock frequency of 100 MHz and avoids the use of the DCM and its additional overhead.

The second bottleneck is encountered when a lookup happens in the trusted state memory component. The covered search space is scanned linearly, getting the entities' states one by one. Subsequently, a comparison is made between the 512-bit identifiers which are to be searched for and the 1280-bit states which are fetched from the trusted state memory component. This bottleneck can be solved by restricting the copied data to the identifier only. Nevertheless, this would still be a bottleneck. It would be more economical to copy the searched identifier to the trusted state and perform the lookup in there. The results on area indicate there is still room to overcome this bottleneck with this solution.

The third bottleneck is similar to the second bottleneck. A lot of data in the shared memory has to copied around to maintain a certain memory map. However, this is not required and could be bypassed. Analogously to the second solution to prevent the second bottleneck, the necessary copies of data in the trusted memory state can be moved to hardware on the trusted state itself.

Another bottleneck might be found in the choice of using Ethernet. On one hand, using a faster communication channel, such as the PCI bus, can reduce the additional delay due to the communication. On the other hand, connecting the trusted hardware through Ethernet allows the sharing of a single hardware module between several log servers.

3.4 Conclusion

This chapter describes a proof-a-concept implementation to strengthen a distributed privacy-preserving log trail scheme. The implementation is realised through the implementation of four cryptographic primitives, together with a dedicated memory core. The strengthening of the logging concept has been proven feasible and solutions for improving the performance have been identified.

Additional threats, introduced by the hardware strengthening, and a brief study on the possible improvements are discussed.

This work has been published in the proceedings of the 15^{th} Euromicro Conference on Digital System Design (DSD) [75]. It also has been published as a chapter in the technical report 'Distributed Privacy-Preserving Log Trails' [64], and led to the master thesis of Ünal and Sallaerts.

Chapter 4

Application: Towards a single-chip, secure, remote FPGA reconfiguration

This chapter describes the second application of this dissertation: a single-chip, secure and remote FPGA reconfiguration. First an introduction presents the application and handles the threat model. In the next section the implementation details are given. Finally, this chapter ends with a discussion of the results and a conclusion.

4.1 Introduction

4.1.1 Situating the application

Bringing electronic devices to the market is an expensive venture. Producing Application Specific Integrated Circuits (ASICs) for these devices is time consuming, which also reflects in the price of the device. This makes designing an ASIC for small volume electronic devices unattractive. Two important differences between FPGAs and custom silicon devices are reconfigurability and price. Although the unit price for an FPGA is higher, this drawback can be compensated by using the reconfigurable nature of the FPGA. Features like remote issue-solving and upgradeability are feasible on FPGAs, which could prolong the life-span of an electronic device as is shown in Figure 4.1. FPGA

configuration updates can be made with physical access to the device, but in the case of electronic devices deployed in the field, it is more efficient to perform updates remotely over a broadband Internet connection. However, remote communication introduces security risks such as the sabotage of a working system or the unauthorised confiscation of IP. Solutions have been presented by FPGA vendors, but they seem insufficient. There is, for example, the attack by Moradi et al. [51], that breaks the encrypted full FPGA configuration using the built-in decryption core. Another example is that the FPGA configuration could not be altered from within the FPGA by means of an encrypted bitstream.

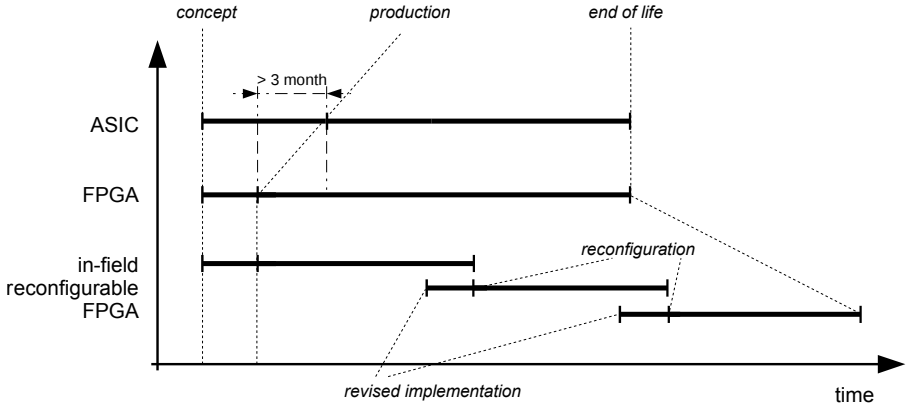

Figure 4.1: The prolonged lifetime of an FPGA driven electronic device

This chapter describes two solutions for the secure, remote reconfiguration of FPGAs which reside in an electronic device with a connection to the Internet. The difference between both solutions lies in the amount of trust that can be put in the electronic device and is explained further in the next section. The security architectures of the solutions do not rely on a Trusted Third Party (TTP) and there is no need for the central configuration instance to keep a list of secret keys for each device in the field. To achieve these features, the FPGA system is based on dynamic partial reconfiguration (DPR). The difference with regular partial reconfiguration is that the static system can continue working, albeit without the use of the functionality which is updated.

In summary, the contributions of this work are the use of entity authentication in the communication, the introduction of remote dynamic partial reconfiguration to obtain an overall online and active partition, and achieving a single-chip solution for remote FPGA reconfiguration.

4.1.2 Threat model

As mentioned above, two different solutions are presented which differ in the amount of trust that can be placed in the device. In the first solution the printed circuit board is considered as trusted, while in the second solution only the FPGA is trusted. The chapter refers to these with Board-is-Trusted (BisT) and FPGA-is-Trusted (FisT), respectively. For both the BisT and FisT solution, the threat model contains four entities, which are shown in Figure 4.2. Attackers can target the communication channel and the device.

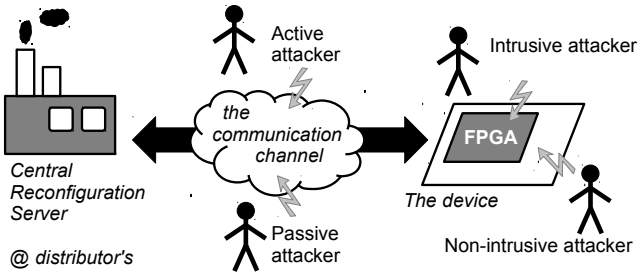

Figure 4.2: The entities in the threat model

The channel

To achieve remote reconfiguration of FPGAs in electronic devices in the field, communication will take place between two entities: the central reconfiguration server and the device that contains the FPGA. The central reconfiguration server is maintained by the distributor of the electronic device and requires a communication channel to the device. It is assumed both passive and active attackers have access to this channel.

A passive attacker can only monitor the channel. Without precautions, eavesdropping the channel can result in IP theft. Passive attackers are assumed to have an inexhaustible supply of storage to store communication sessions, which allows the attacker to compare intercepted messages with every message ever sent. Finding patterns in communications could leak information.

An active attacker cannot only monitor the channel, but can also alter or introduce the information that is transferred over the channel. Being able to alter the channel could lead to spoofing attacks or man-in-the-middle attacks. In the former case, active attackers can alter the content of a message, that is sent by an entity. In the latter, the attacker impersonates both entities, while these entities believe they are communicating with each other. Both attacks

pose a threat to the participating entities. Without precautions, unauthorised bitstream transfers can result in the malfunctioning of the device.

The device

The device containing the FPGA can be threatened by non-intrusive and by intrusive attackers, who both have physical access to the device. The former monitor pins, circuits, or power consumption as side-channels in search of leaking information. The latter try to impede the correct behaviour by manipulating the device. This can be done by re-routing wires, altering the supply voltage, by breaking the housing of the FPGA and interfering with the silicon, or by other destructive attacks.

For the BisT solution, only the threat of non-intrusive attackers, performing side-channel analysis attacks based on the observation of power traces, is taken into consideration. In the FisT solution an intrusive attacker who can manipulate wiring on the printed circuit board, is also considered.

4.1.3 A summary of the solutions

In the majority of the DPR designs, the static partition covers the largest part of the FPGA, while the reconfigurable partitions are kept small. In contrast to this general approach, the presented solutions place the main functionality of the FPGA in a single reconfigurable partition. The static partition of the FPGA contains the overhead that is required for allowing secure and remote communication and for reconfiguration. For this reason it is important to keep the static partition as small as possible, leaving as much of the reconfigurable area available for the main application.

The single reconfigurable partition can be reconfigured with a partial bitstream, that is to be generated at the central reconfiguration server. This latter entity sends the partial bitstream containing the updated behaviour to the FPGA over the Internet. In order to make the proposed solutions address the threat to communications, a secure communication protocol is necessary. The required protocol should provide three cryptographic principals, as defined in Section 1.3.1: data confidentiality, data authentication, and mutual entity authentication. With such a protocol a cryptographic key can be established that can be used to encrypt and authenticate the new bitstream.

When the bitstream securely arrives at the FPGA, the data integrity should be verified before reconfiguring the reconfigurable partition. An additional challenge is the storage of the partial bitstream. In the BisT solution external

memory is used to store the incoming bitstream, but for the FisT solution the bitstream should not leave the FPGA chip. Data compression should be used to implement the FisT solution.

An additional benefit of the proposed solutions is the reduced downtime of the main applications. Because the partial bitstream is already near or at the FPGA, the communication time is strongly reduced.

4.2 Implementation

4.2.1 Related work

This section summarises related work that has been done on reconfiguration of FPGAs, together with a summary of the work done on data compression with respect to bitstreams.

Work on secure FPGA reconfiguration

The work of Castillo et al. [8] describes a system for a remote self-reconfiguring FPGA. They offer data confidentiality and data authentication. The former is guaranteed by encrypting the bitstream with the DES algorithm, while the latter is done with a MD5 hash. It is also mentioned that the data source should be authenticated using public-key based cryptography, but this was not implemented. In contrast, both the BisT and FisT solutions use stronger algorithms and provide an implementation for the public-key approach.

Gonzalez et al. make a comparison between two softcore processors: the MicroBlaze and the LEON2 [24], which are used in a self-reconfigurable system in which an implementation of a cryptographic algorithm can be changed. Our work focuses on general FPGA reconfiguration rather than on IP cores. In both the BisT and FisT solutions the reconfigurable partition is a standalone implementation, which has no functional connections to the static partition.

In [20], Drimer and Kuhn present a protocol for secure, remote updates of FPGA configurations. Their work provides data confidentiality, data authentication and freshness of bitstreams. They use a new protocol, in contrast to the protocol used in our solution which has been around for more than twenty-five years and is still not broken. In their work, newly communicated bitstreams are stored in non-volatile memory, either already decrypted or still encrypted. The former is to be used on low-end devices (that do not provide decryption during configuration), which allows attackers to eavesdrop the decrypted bitstream.

The latter is to be used on high-end devices (that provide decryption during configuration), which allows attackers to eavesdrop the encrypted bitstream by using advanced techniques as presented by Moradi et al. in [51]. In the BisT solution the bitstream never leaves the FPGA in plain text, while in the FisT solution the bitstream never leaves the FPGA at all. Moreover, the incoming bitstream reconfigures the FPGA through the Internal Configuration Access Port (ICAP) thus avoiding the vulnerability, exploited by Moradi et al. [51].

SeReCon, by Kepa et al. [32], is a secure reconfiguration controller for self-reconfigurable systems. The work focuses on updating and interchanging IP cores. Even IP cores from third parties can be accepted if their fitter component detects no issues. However, with SeReCon the reconfigurable parts of the FPGA are reduced to IP cores which fit into the design. The overhead of their system is also rather substantial: the SeReCon component uses two MicroBlaze softcores with accompanying buses. The transfer of the IP core bitstreams is not described, but the only communication components available in their system are a JTAG port and a serial port, which both require the physical presence of a technician to update IP cores.

In [16], a secure protocol and its implementation, for remote bitstream updates is presented by Devic et al.. They focus on preventing replay attacks. To achieve this, they introduce the use of tags on bitstreams, which are counters on the succession of the bitstreams. In more recent work, Devic et al. present the use of a secure protocol 'SecURe DPR' to prevent bitstream spoofing and replay attacks for dynamic partial reconfiguration [17]. Their threat model does not consider side-channel analysis attacks, as introduced in [38, 39]. Both in the BisT and FisT solutions, spoofing attacks and replay attacks are countered, while measures for power-oriented side-channel analysis resistance are also taken up to a certain level.

Table 4.4 on page 99 summarises a comparison between our work and other published, related work. Only related work which targets a broader reconfiguration than IP cores, is taken up in this comparison.

Work on bitstream data compression

In [27], Pan et al. present a new technique: Difference Vector compression, which gains data compression by sending only the differences between a configuration frame and a beneficiary frame. They extend this technique by reading back frames from the current configuration, which are used as a reference to build the Difference Vector.

Koch et al., in [36], use three techniques: run-length encoding (RLE), Lempel-Ziv algorithms and Huffman compression. They compared these techniques on a set of benchmark bitstreams both on compression ratio and resource cost of the decompresser.

Haiyun and Shurong have implemented in [26] the Lempel–Ziv–Welch (LZW) algorithm [15]. Their work achieves a compression ratio of 43.69% for partial bitstreams.

The work of Ştefan and Coţofană [13] focuses on bitstreams for Xilinx Virtex-4 devices. They use several techniques and combinations of techniques to achieve bitstream compression and they discuss the accompanying hardware implementations to perform the decompression. They draw two main conclusions: 1) the internal organisation of bitstreams is likely to change between FPGA families and 2) synthesis tools produce bitstreams with the ratio of '0' and '1' symbols as the main source of redundancy. The latter turns the focus on the simpler compression methods.

A combination of run-length encoding and bitmask-based compression is used in the work of Qin et al. [65]. They outperform other compression techniques, while maintaining high speed decompression.

4.2.2 Cryptographic protocol and algorithms

The first step in the communication consists of the two entities agreeing on a secret, only known to those two entities, by using a cryptographic protocol. This secret, or cryptographic key, will enable the secure transmission of the (partial) bitstream in the communication later on. The protocol family candidates for agreeing on a shared secret, are hierarchically summarised in Figure 4.3.

To choose an appropriate protocol, the hierarchy of Figure 4.3 needs to be descended.

Long-term or session key

Figure 4.3 illustrates that there are two possible keys that can be agreed upon: a long-term key and a session key. As explained in [45], the use of a session key can be motivated by the following arguments:

- to limit the amount of available cipher texts (under a fixed key) for cryptanalytic attacks;

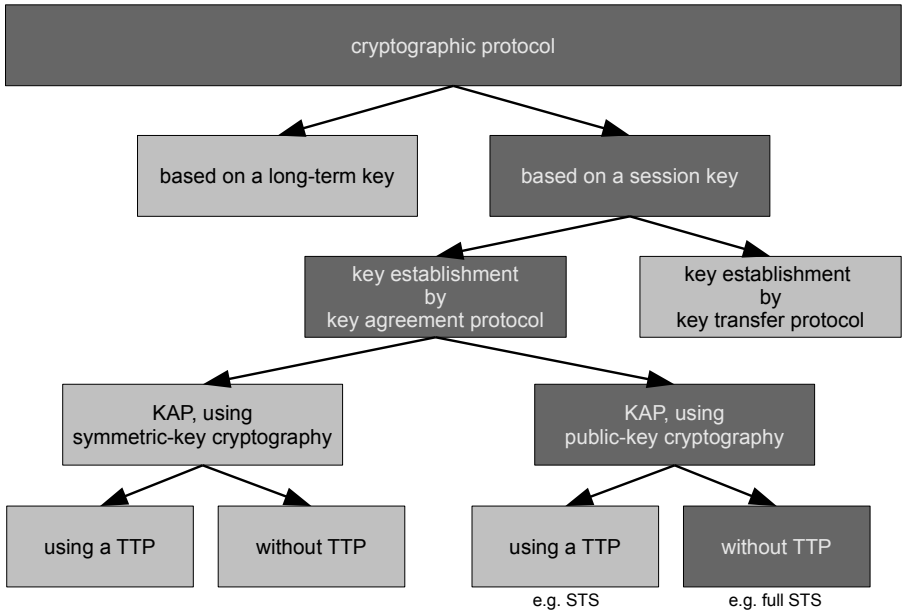

Figure 4.3: The cryptographic protocol family candidates, where KAP is a key agreement protocol and TTP is a trusted third party. The dark grey boxes indicate the choices which were made.

- to limit the exposure, with respect to both time period and quantity of data, in the event of a (session) key compromise;
- to avoid the long-term storage of a large number of distinct secret keys (in the case where one entity communicates with a large number of entities), by creating keys only when actually required;
- to create independence across communication sessions or applications.

Key establishment by a key agreement or key transfer protocol

In this application the session key is to be agreed on, every time a communication session is initiated. To be more precise, a key establishment protocol runs between the central reconfiguration server and the device in the field containing the FPGA, prior to every communication session. Key establishment protocols can roughly be divided into key transport protocols and key agreement protocols. In case of the former, there is one entity that determines the secret and securely transfers it to the other entity. In the latter type, the shared secret is generated

by both entities so no single entity can predetermine the resulting secret. The freshness of the session key is guaranteed by the fact that every entity actively contributes to generate the session key.

Key agreement through symmetric-key or public-key cryptography

There are mainly two techniques to implement a key agreement protocol. One is based on symmetric-key cryptography and the other one is based on public-key cryptography. In this work a public-key protocol is used. The reason for doing so is key management. Using symmetric-key cryptography, a key has to be stored centrally for every device that leaves the factory floor. Whereas using public-key cryptography, the use of public keys and certificates for those keys, makes the central storage of a list of private keys unnecessary. Such a list would be an attractive target for attackers because they could gain access to every device in the field, by only attacking a single point. Another reason for choosing public-key cryptography is the possibility to expand towards a public-key infrastructure, which would allow multiple and distributed reconfiguration servers. Note that the choice for a public-key based protocol for key agreement does not mean public-key cryptography is used for the encryption/decryption of the reconfiguration data.

Public-key cryptography with or without trusted third party

Further, some public-key protocols make use of a trusted third party (TTP), while others do not. The TTP is often also referred to as the certification authority or the authentication server. The TTP could act in the role of a registration authority, a key generator, a key server, and other roles or combinations of roles. In the presented setting, each electronic device originates from a site that is trusted by the distributor. Here, the certificate $Cert_B$, which is explained later, and the public key A of the central reconfiguration server are stored on the device. It is assumed that the distributor prefers to trust as little outsiders as possible. Hence, a public-key protocol without TTP is opted for.

The above specifications and properties result in the choice of the Station-To-Station (STS) protocol [45]. To fulfil the requirement of not using a TTP to verify the public keys, the full STS protocol is used. The STS protocol offers mutual entity authentication, mutual explicit key authentication, mutual key confirmation and perfect forward secrecy. The latter is defined in [45] as: a protocol is said to have perfect forward secrecy, if compromise of long-term keys does not compromise past session keys.

Table 4.1 shows the full STS protocol using some custom notations. A digital signature with a private key x on a message M is noted: $S_x(M)$. Encrypting a message M with a symmetric key x is noted: $E_x[M]$, while $D_x[M]$ represents the according decryption of a message. Finally it is pointed out that in a public-key pair (a, A) the lower case letter is the private key, while the corresponding upper case letter is the public key.

The difference between STS and full STS is that the latter uses public key certificates while the former does not. These public key certificates are digital signatures, signed with the private key of the central reconfiguration server, on the public key of the other entity. This allows each entity to verify the public key of the other entity, by using the public key of the central reconfiguration server. Table 4.1 also shows that the elliptic curve variant of STS is used. Digital signatures are generated and verified using the Elliptic Curve Digital Signature Algorithm (ECDSA) [60]. The reason for using elliptic curve cryptography is key size. As can be read in [21], modular exponentiation and elliptic curve point multiplication have an equivalent strength while the latter uses much smaller keys than the former. This results in a smaller implementation.

After the completion of the full STS protocol, the shared secret is a point on an elliptic curve, consisting of an X and Y coordinate which both are 256 bits in size. Both coordinates are hashed to result in a 256-bit session key from which two keys are obtained: key_1 and key_2, being the 128 left most and right most bits respectively. The first key is used as key for the encryption of the messages, while the second key is used to calculate the MAC. The 128 right most bits of the X coordinate of the secret point on the elliptic curve are used as initialisation vector for the symmetric-key cipher.

The full STS protocol, as shown in Table 4.1, uses a specific set of operations. This set can be summarised as follows:

- random number generation
- elliptic curve point operations
- symmetric-key encryption/decryption
- generation and verification of a MAC
- generation and verification of digital signatures

In addition to this set of operations, a hash function is used to generate key_1 and key_2. This requirement is also needed for the generation and verification of the digital signatures and is therefore not considered too costly.

Table 4.1: The full Station-To-Station protocol

Central reconfiguration server		**Electronic Device with FPGA**
key pair (a, A)		key pair (b, B)
generator P		generator P
$Cert_A = S_a(A)$		$Cert_B = S_a(B)$
		A
choose k_1		
$Q_1 = k_1 * P$		
$X = Q_1$	X	
	$\rightarrow$	
		choose k_2
		$Q_2 = k_2 * P$
		$K = k_2 * Q_1$
		$key_1 \mid\mid key_2 = H(K_x, K_y)$
		$S_b(Q_2 \mid\mid Q_1)$
		$E_{key_1}[S_b(Q_2 \mid\mid Q_1)]$
		$Y = Q_2 \mid\mid B \mid\mid Cert_B \mid\mid E_{key_1}[S_b(Q_2 \mid\mid Q_1)]$
	Y	
	$\leftarrow$	
$K = k_1 * Q_2$		
$key_1 \mid\mid key_2 = H(K_x, K_y)$		
$D_{key_1}[E_{key_1}[S_b(Q_2 \mid\mid Q_1)]]$		
verify $Cert_B$		
verify $S_b(Q_2 \mid\mid Q_1)$, using B		
$S_a(Q_1 \mid\mid Q_2)$		
$E_{key_1}[S_a(Q_1 \mid\mid Q_2)]$		
$Z = Cert_A \mid\mid E_{key_1}[S_a(Q_1 \mid\mid Q_2)]$	Z	
	$\rightarrow$	
		$D_{key_1}[E_{key_1}[S_a(Q_1 \mid\mid Q_2)]]$
		verify $Cert_A$
		verify $S_a(Q_1 \mid\mid Q_2)$, using A

The first four operations can be implemented using the implementations from Chapter 2. The fifth operation can be realised by using these implementations as well.

Figure 4.4 shows a graphical representation of the hierarchical levels of the required operations in ECDSA: random number generation, elliptic curve operations and the calculation of a hash value. The elliptic curve operations are performed on the same datapath as the elliptic curve operations in the other

steps of the STS protocol. Finally, the standardised Secure Hashing Algorithm (SHA-256) with a digest length of 256 bits is part of the current standard and is hence an acceptable choice [59].

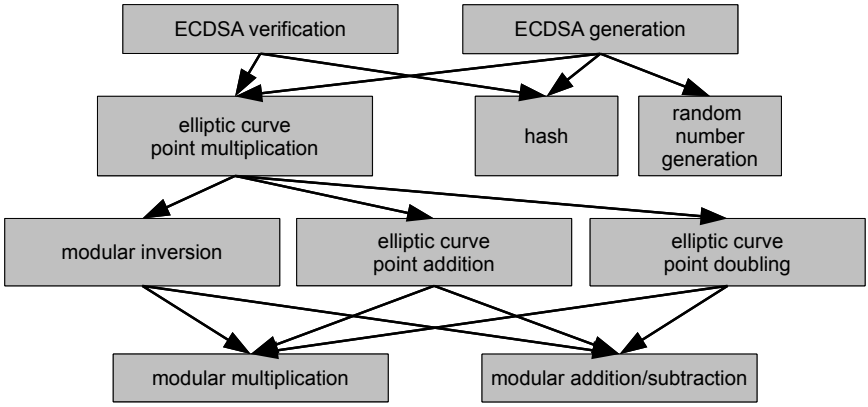

Figure 4.4: The hierarchical levels of the required operations in ECDSA, according to [60]

4.2.3 Architecture

The top level architecture of the BisT and FisT solutions is shown in Figure 4.5.

The outer rectangle of Figure 4.5 represents the printed circuit board which contains the FPGA and the external memory for implementing the BisT solution. The FPGA is divided into a reconfigurable and static partition. While the former is dedicated for the main application, the static partition holds the required resources for the communication and the reconfiguration. The controlling component in the static partition is a MicroBlaze softcore CPU, which uses a single dual-ported BRAM which serves as data and instruction memory (IMEM and DMEM) each with their own bus. The processor local bus (PLB) is used to interconnect five peripherals: an Ethernetcore, the ICAP, the cryptocore, either an external memory controller or a (set of) BRAM(s) and a placeholder component for the reconfigurable partition. Whether the external memory or the on-chip memory is used, depends on the implemented solution. For the BisT solution the external memory controller is used, while the FisT solution requires one or more BRAMs.

The 'cryptocore' is a custom-built peripheral which contains the implementations of the four required cryptographic primitives. These four components are

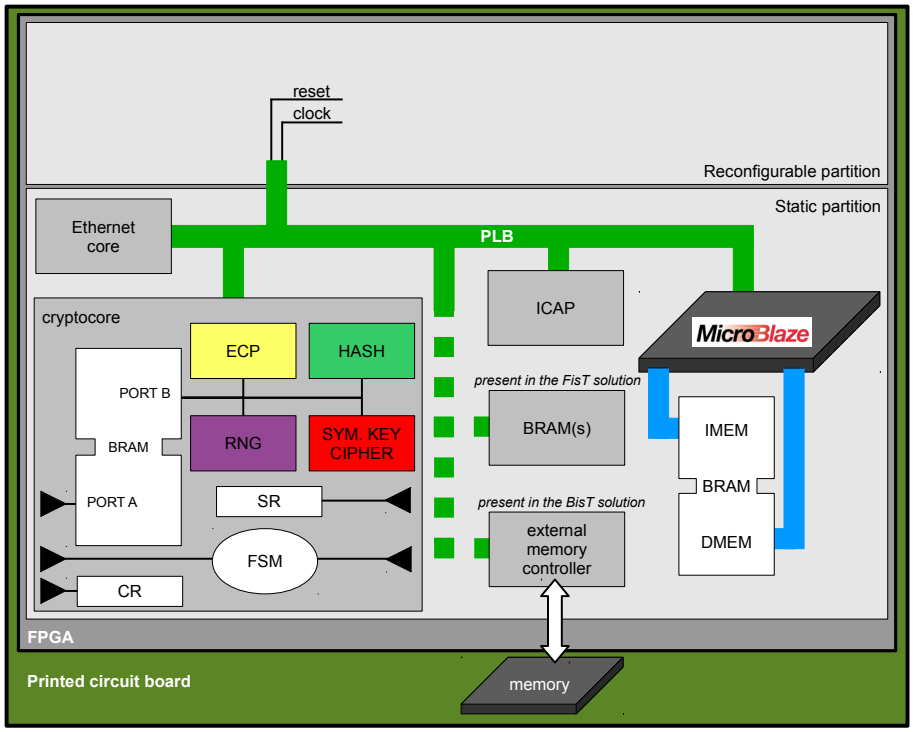

Figure 4.5: The top level architecture of the BisT and FisT solutions

interconnected through a shared BRAM and are coordinated by an FSM which is interfaced through a command register (CR) and status register (SR). These registers can be written or read through the PLB bus.

Figure 4.5 shows a connection of the reconfigurable partition to the PLB bus. There are no functional connections through this bus. The placeholder component could indicate the contrary, but only the reset and clock signals come in through this 'connection' because their input/output buffers to the physical connections are taken up by the static system. The need for such a placeholder component comes from the implied tool flow that comes with PlanAhead, the vendor tool for implementing partial reconfiguration designs.

4.2.4 Software on the CPU

A short explanation on getting software, which has to run on the MicroBlaze, into a bitstream is given in Section 3.2.2.

The MicroBlaze communicates with the central reconfiguration server using an Ethernet connection with the UDP/IP protocol. Reasons for choosing the UDP protocol over the TCP protocol are given in Section 3.2. The software running on the MicroBlaze starts by initialising the Ethernet softcore and initialising constant parameters which are mainly used for program offsets, network settings, padding messages for the HASH/HMAC component and the keys used in the scheme. After the initialisation the MicroBlaze keeps polling the Ethernet component for incoming instructions, following a flowchart as depicted in Figure 4.6.

The MicroBlaze first handles possible requests using the address resolution protocol to find the Media Access Control (MAC) address of the network interface card with a certain IP address. When an IPv4 packet destined for the FPGA arrives, the requested action is looked for in the first payload byte. This action can be a message of either the key establishment protocol or the reconfiguration protocol.

Upon receiving the first message of the STS protocol, the MicroBlaze handles the next steps of the full STS protocol. If a single check in the protocol fails, the MicroBlaze interrupts the remaining steps and waits for the first message of the STS protocol again. When the third message of the STS protocol arrives, the MicroBlaze finishes the protocol and sends a MAC-ed and encrypted acknowledgement.

Depending on the size of the reconfigurable partition, the size of the partial bitstream also varies. The maximum payload which can be sent in a single, default UDP frame is 1 472[8] bytes. Since the aim in this application is to get the reconfigurable partition as large as possible, it is assumed that the partial bitstream will not fit in a single UDP frame. The simple reconfiguration protocol organises this administration.

The partial bitstream will have to be divided into multiple *chunks*. A MAC is appended to every bitstream chunk using key_2. This bitstream chunk and its MAC are encrypted using key_1. The reconfiguration protocol handles three types of requests:

1. receive an encrypted bitstream chunk with MAC
2. check the validity of the received bitstream
3. reconfigure the FPGA

[8]The 1 500 bytes maximum payload of an Ethernet frame minus the IP header and UDP header of respectively 20 and 8 bytes results in a 1 472 bytes payload.

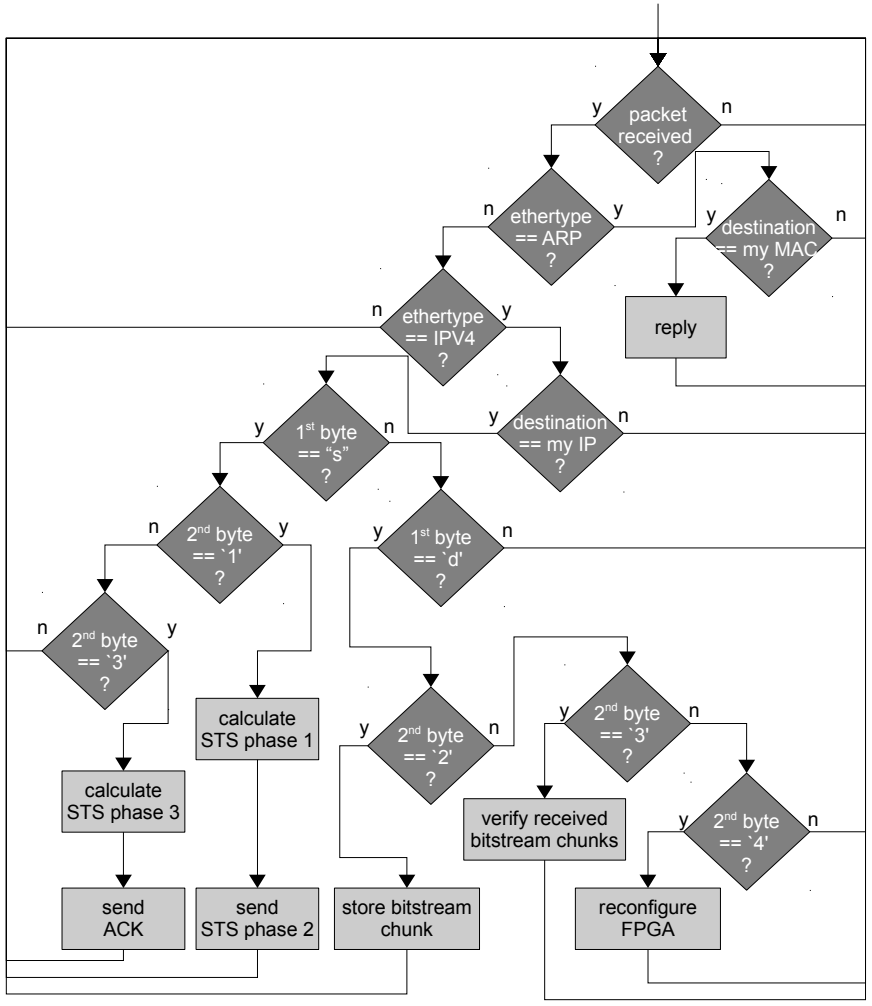

Figure 4.6: Flowchart of the software running on the MicroBlaze after initialisation

The first type of requests accepts an encrypted bitstream chunk together with its MAC and stores it in memory on an address based on the order number. For the BisT solution this is in external memory, while the FisT solution stores the payload on-chip. The reason for using the order mechanism is that the UDP protocol cannot guarantee that the chunks arrive in the same order as they are sent by the central reconfiguration server. The possible re-ordering has to be done by the receiver.

When all chunks are sent, the central reconfiguration server can ask the MicroBlaze to check the validity of the received bitstream, which is the second type of request. Upon such a request, the distributor also has to send a MAC on the entire partial bitstream. The MicroBlaze will scan the memory, decrypting every encrypted chunk on the FPGA and verifying every MAC on a chunk. Subsequently the MAC on the entire bitstream is verified. If this latter verification succeeds, it is certain that 1) every chunk has been received; 2) the correct order of the chunks has been restored; 3) the received partial bitstream is unaltered; and 4) the partial bitstream was sent by the central reconfiguration server and is therefore trusted. Only upon a correct verification of the MAC on the entire partial bitstream a flag in the MicroBlaze's data memory is set. This flag signals that a correct partial bitstream is available. After setting this flag, every MAC on a single bitstream chunk is removed to obtain a non-segmented partial bitstream. Additionally for the FisT solution, every chunk is decrypted resulting in a continuous decrypted compressed bitstream. The evolution of the data in memory is visualised in Figure 4.7.

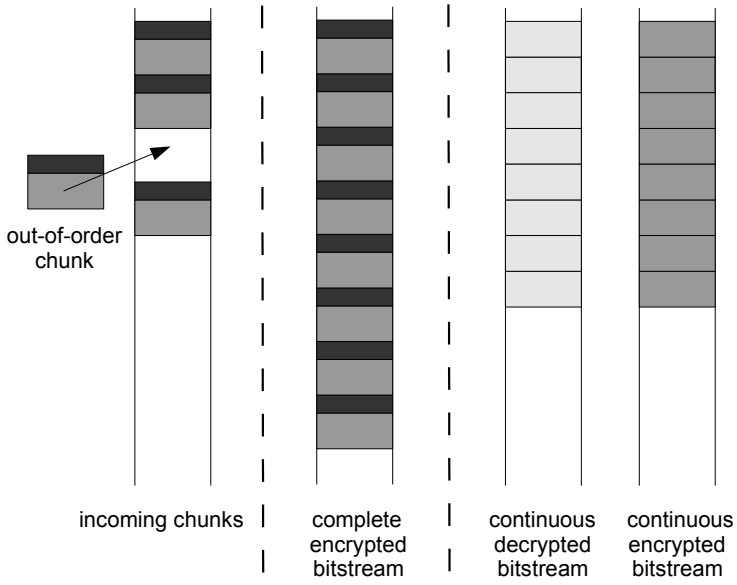

Figure 4.7: The evolution of the memory storing the bitstream chunks

The third type of requests accepted by the MicroBlaze is the writing of the partial bitstream to the ICAP. However, this only happens if the internal flag, mentioned above, is set. In the BisT solution every chunk has to be decrypted on the FPGA before moving it to the ICAP. In the FisT solution the data decompression has to be performed prior to moving the chunk to the ICAP. After

writing the partial bitstream to the ICAP, the behaviour of the reconfigurable partition is updated.

Up until this point the BisT and FisT solutions are not too different from each other. The remainder of this section will be considered for both solutions separately.

Board-is-Trusted

Given that the maximum payload of a single UDP frame is 1472 bytes, one can determine the chunk size. There are two[9] additional bytes to store the order number and 32 additional bytes for the MAC. This leaves 1438 available bytes for actual chunk data. This fits 89 AES blocks of 128 bits, which results in a chunk size of 1424 bytes. The packet payload and its generation procedure for the BisT solution are visualised in Figure 4.8.

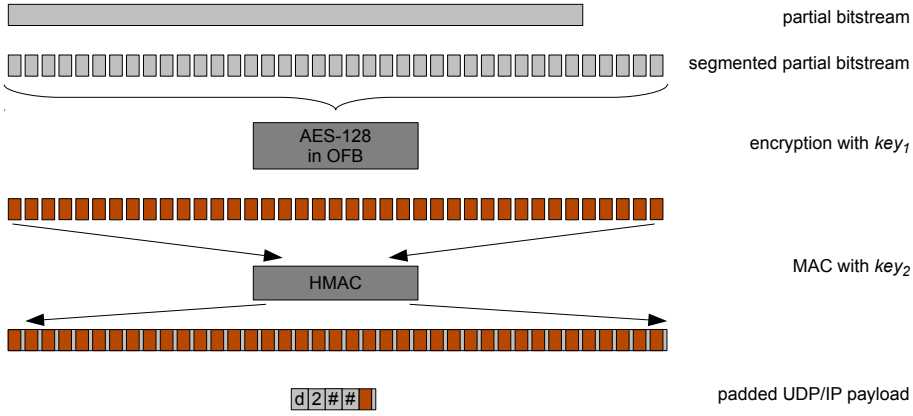

Figure 4.8: The UDP/IP packet payload and its generation procedure used in the BisT solution

FPGA-is-Trusted

For the FisT solution the available storage is scarce. To be able to store the partial bitstream on the on-chip memory, data compression is required. Note that most research on bitstream compression is performed to reduce the configuration time of the FPGA, while the goal in the FisT solution is to avoid the need for external memory.

[9] A full bitstream for the XC5VFX70T is 3'378'269 bytes which fits 2'373 (0x0945) chunks.

There exist many compression algorithms that offer lossless data compression. It needs no further explanation that lossy compression algorithms are unsuitable. For clarity it is noted that the compression ratio (CR) is defined as:

$$CR = \frac{size(compressed\,bitstream)}{size(original\,bitstream)}$$

Many algorithms are based on: Lempel-Ziv algorithms, Huffman encoding and/or run-length encoding. Lempel-Ziv-based algorithms focus on substitutions of repeating occurrences, which are maintained in a dictionary [84]. Huffman coding uses short words to represent frequently occurring words, while infrequently occurring words are represented by longer words. For decompression the Huffman tree needs to be reconstructed. This is considered to be costly [13, 36]. Run-length encoding is a very simple technique, that compresses the data by indicating consecutive repetitions in the data. The compressed output will exist of pairs of (w, l) describing the content of the input. The w field defines the word, while the l field gives the number of repetitions. All three mentioned algorithms have a fixed word-size in a given compression, but this word-size is adjustable.

Given that the maximum payload of a single UDP frame is 1472 bytes, one can determine the chunk size. There are two additional bytes to store the order number, analogous to the BisT solution, and 32 additional bytes for the MAC. This leaves 1438 available bytes for actual chunk data. One byte is reserved for the length of w and two bytes are reserved for the length of l. These three bytes serve as metadata for the compression of the chunk. This fits 89 AES blocks of 128 bits, which results in a chunk size of 1424 bytes. The packet payload and its generation procedure for the FisT solution are visualised in Figure 4.9.

4.2.5 Implementation results

Area

Table 4.2 summarises the required resources for the BisT and FisT solution, on a Xilinx ML507 development board containing a Xilinx XC5VFX70T FPGA. The tools used to quantify the required resources are part of the Xilinx ISE Design Suite, version 14. Figure 4.12 shows the floorplan of the FPGA with the placed and routed designs, for the BisT (left) and FisT (right) solution. A graphical representation of the relative resource usage is shown in Figure 4.10 and Figure 4.11.

The cryptocore in its entirety uses around 25% of the targeted FPGA. The efforts for making a small implementation of the cryptographic components pay

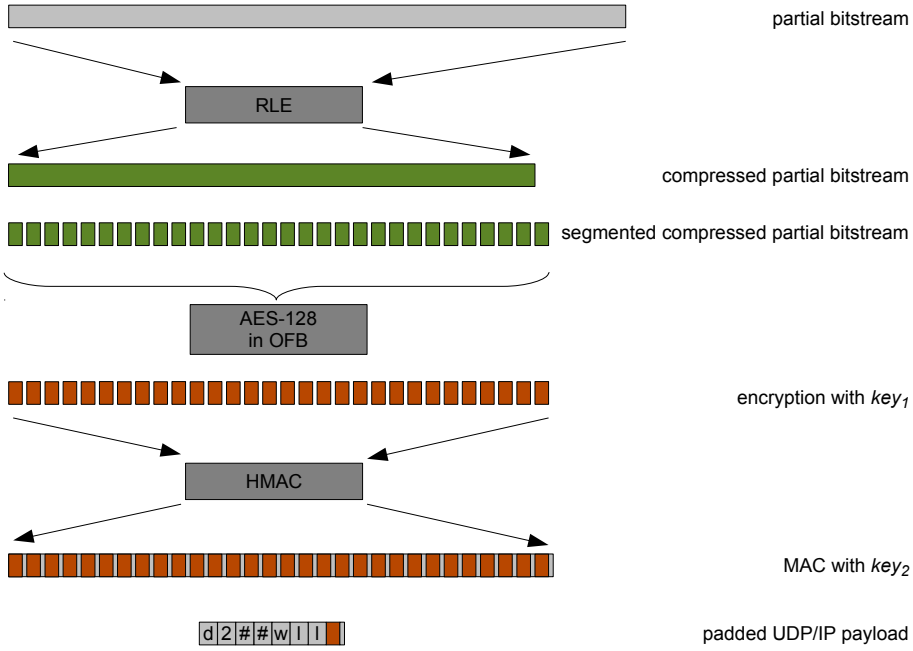

Figure 4.9: The UDP/IP packet payload and its generation procedure used in the FisT solution

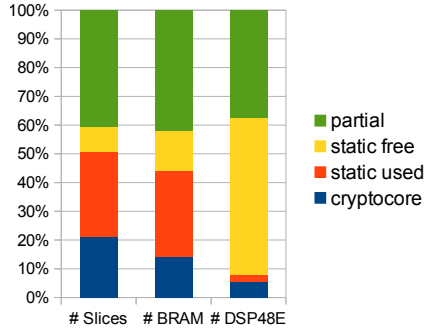

Figure 4.10: Relative resource usage of the BisT solution

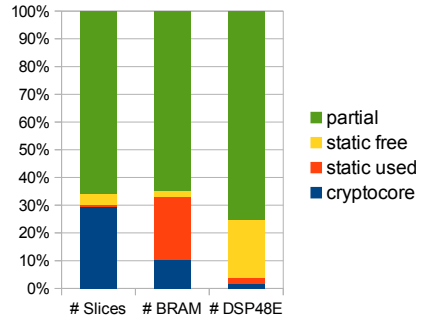

Figure 4.11: Relative resource usage of the FisT solution

off, but Table 4.2, Figure 4.10 and Figure 4.11 also indicate that the cost of the cryptocore is still relatively high. It is pointed out that the optimisation by the design tools, on the whole of the static partition, forces a more efficient usage of the resources.

Table 4.2: Resource usage of the cryptocore and both partitions

component	Number of Slices	BRAM	DSP48E
BisT: Cryptocore[1]	2 378	21	7
BisT: Static partition	5 702	65	10
BisT: Available in Static partition	6 640	86	80
BisT: Available in Reconfigurable partition	4 560	62	48
FisT: Cryptocore[2]	3 305	15	2
FisT: Static partition	3 392	49	5
FisT: Available in Static partition	3 840	52	32
FisT: Available in Reconfigurable partition	7 360	96	96
Available in XC5VLX20T	3 120	26	24
Available in XC5VFX70T	11 200	148	128
Available in XC5VLX330T	51 840	324	192

[1]: contains impl10 + impl20 + impl30 + impl40, from Table 2.13
[2]: contains impl10 + impl20 + impl31 + impl41, from Table 2.13

The additional resources occupied by the static partition contain the MicroBlaze with its peripherals. For the BisT solution, this includes the external memory component while the FisT solution contains a single BRAM and its interface.The software for both solutions fits in the single BRAM used by the MicroBlaze.

From Table 4.2 it is clear that the static partition has more reconfigurable resources available than is required. How to make the segmentation in the static and the reconfigurable partition is mostly determined by the size of the static partition's content, but for the FisT solution, also depends on the size of the main application. If the size of the main application is larger, additional storage to save the new configuration is required. This makes determining where the border between static and reconfigurable partitions is to be placed, a difficult and application-specific task.

Figure 4.12 shows the placed-and-routed full bitstreams for the BisT and FisT solution. The marked boxes on both floorplans, indicate the reconfigurable partitions of the FPGA in both solutions. As is clear from the figure, the constraints in both solutions differ significantly. This has no technical reason,

but illustrates how the segmentation can vary. The density in the static partition of the BisT solution is visually lower than in the FisT solution. The results in Table 4.2 numerically support this by indicating that the static partition of the BisT solution has more unused resources than the FisT solution. The routing through the reconfigurable partitions are the reset and clock signals and routing from and to inputs and outputs.

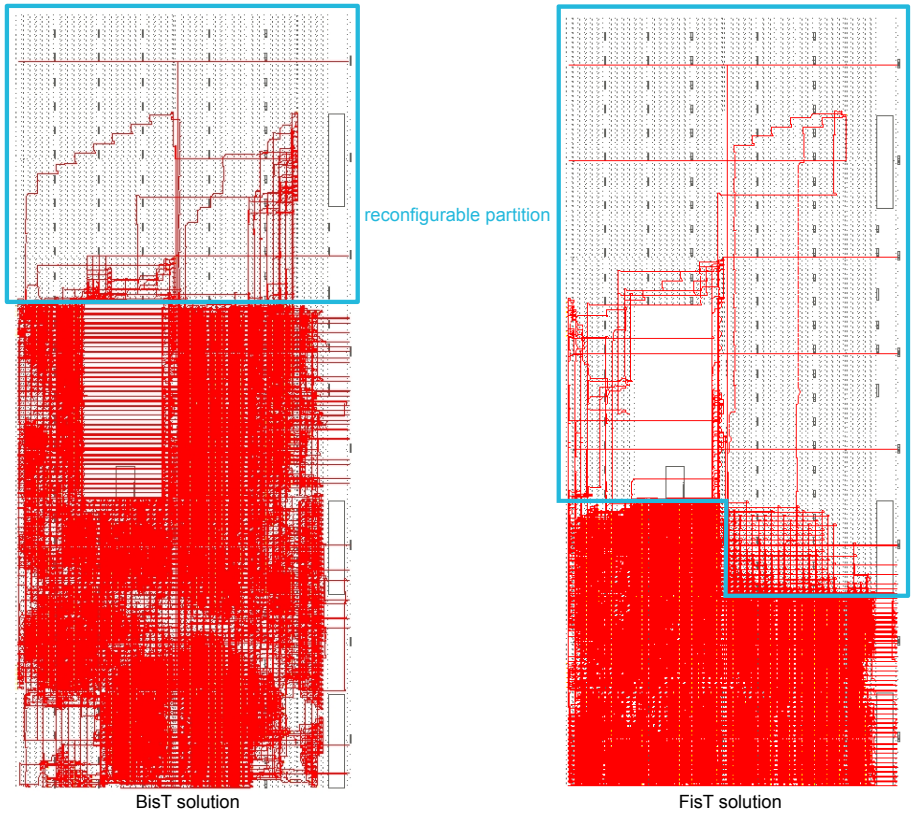

Figure 4.12: The placed-and-routed full bitstreams for the BisT (left) and FisT (right) solutions

Finally, Figure 4.13 shows the resource usage of both the BisT and FisT implementation on recent Xilinx FPGAs. These results compare the number of occupied resources to the available resources without taking in account the possible optimisations that come with these newer devices or with more recent development software.

The substantial overhead of both solutions is minimised on the more recent and larger FPGAs.

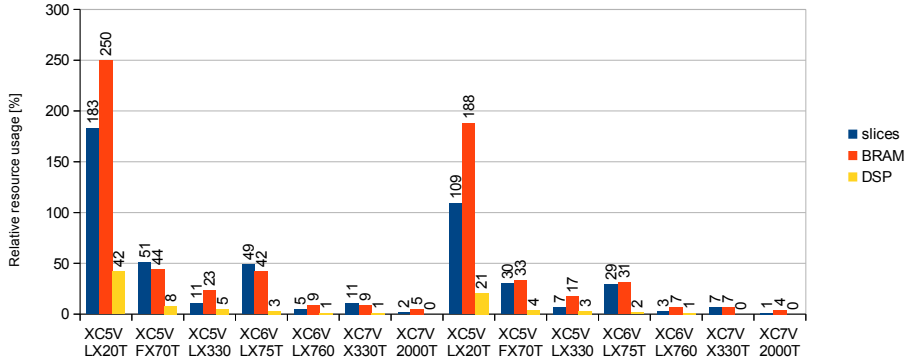

Figure 4.13: The relative resource usage of both solutions on recent Xilinx FPGAs

Speed

The focus of the complete design for both solutions is on minimal area rather than on maximal speed. Nevertheless, to report on the timing of the design, a small led visualisation application is implemented as main application.

For the BisT solution, the full bitstream is 3'378'281 bytes in size, while the partial bitstream is 62'357 bytes. The partial bitstream which is sent over the network requires padding for cryptographic operations and encapsulation for the network transfer. This results in a partial bitstream of 64'480 bytes.

Table 4.3 on page 98 shows the results of timing of the different types of FPGA reconfigurations. These measurements were taken on a local lab network. When reconfiguring the FPGA using a JTAG connection, the shares of duration of the communication and the reconfiguration are not available. This table indicates that the overhead of the cryptographic protocol is equal to the data transfer time for small bitstreams. This ratio will improve when bitstreams get larger.

The timing results mentioned above can vary heavily in other setups and designs. The first and most important reason is that the size of the partial bitstream is proportional to the size of the reconfigurable partition. The second reason is that the latency of the network has a substantial influence on the duration of the communication.

4.3 Results

4.3.1 Security

The cryptographic protocol, as described in Section 4.2.2, uses a session key for every new communication session. During key establishment, both entities can authenticate each other by using the certificate. Using the session key guarantees that the transfer of the partial bitstream is confidential because of AES in OFB mode and that the FPGA on the device is certain of the authenticity of the data because of the HMAC. Up to this point, the security is ensured by the strength of the algorithms chosen in Section 4.2.2 and by protecting the cryptographic keys, both the private key and the session keys from active and passive attackers as well as from intrusive and non-intrusive attackers.

At the moment of writing, the private key is stored in the pass-through memory, on a default address. As proof-of-concept this can work, but it is not a design practice as there are two main issues.

A first issue is that the bitstream that brings up the static system, contains the private key. By eavesdropping this initial configuration, the intrusive and non-intrusive attacker could learn the private key. There is a possibility of encrypting the initial bitstream, but this does not solve the problem. The first reason is a published attack on encrypted bitstreams by Moradi et al. [51]. A second reason is that this would only move the issue of storing the private key of the FPGA to storing the symmetric key, used for the encrypted, initial configuration.

The second issue with storing the private key in the pass-through memory is the fact that this key resides at a constant address in a well defined set of BRAMs. This could allow fault injection attacks, as introduced by Carreira et al. in [7].

A solution for storing the private key could be obtained from Physical Unclonable Functions (PUFs) as introduced by Gassend et al. [23]. These functions can collect unique information, based on physical properties of a specific device. This information can be used to generate the private key, as presented by Maes et al. in [42].

As discussed in the threat model, the intrusive attacker can also attack the device physically. A first method for attacking the system can be very simple. Pulling the power plug off the board, removing the UTP cable, and other physical interventions result in a denial of service (DOS) attack. Since protecting against such attacks needs to be done in the physical world, these attacks fall outside of the scope of this work. Another type of DOS could be to flood the MicroBlaze

with obsolete requests. This type of DOS attack should be held off at the network level.

A second method is for intrusive and non-intrusive attackers to perform a timing attack [38]. This type of attack tries to gain information from measuring time during specific operations of the device. In Table 4.1 a timing attack could be used to try to steal the random number k_2 from the calculation $Q_2 = k_2 * P$. With k_2 obtained from a timing attack and Q_1 sent in plain text, key K can be calculated and so the session key could get stolen, thus breaking the scheme. To protect against timing attacks the implemented method for calculating the elliptic curve point multiplication is the Montgomery ladder instead of a faster, straightforward approach. Doing so results in a longer but time-constant path, independent of k_2.

As a third method, intrusive and non-intrusive attackers can do side-channel analysis attacks. Only side-channel analysis attacks on power traces are discussed here, since countermeasures against other types of side-channel analysis attacks have not been implemented. A SPA attack on power traces is not applicable because of the use of the Montgomery ladder (explained in Section 1.3.4). The threat for differential power analysis attacks on the ECP, calculating Q_2, is minimal because $k_2 * P$ is performed with a random scalar with every iteration of the protocol.

The session key is the result of key material chosen at the distributor site and at the site of the electronic device. This ensures the freshness of the session key, preventing replay attacks. With the incoming partial bitstreams being encrypted and having a MAC, both with the freshly generated session key, and with the presence of the internal flag of having received a complete and valid bitstream, spoofing and replay attacks are countered.

In [5], Blake-Wilson and Menezes have published unknown-key-share attacks on the STS protocol. This attack can result in a situation where the central reconfiguration and the electronic device do share a generated session key, but the electronic device is convinced it shares a key with an unknown entity. This attack can be prevented by moving the certificates in the second and third pass of the full STS protocol inside the encryption. This can be achieved at almost no additional overhead.

4.3.2 Reconfiguration protocol

It is pointed out that the proposed protocol might be slow. However, this is not considered an issue because the main application can continue working during this communication. The actual downtime of the main application is limited

to the time window it takes to read the partial bitstream from memory and write it to the ICAP. This downtime is proportional to the size of the partial bitstream and therefore proportional to the size of the reconfigurable partition.

The reconfiguration protocol can be used as described above. Nevertheless, improvements can be applied. Four important improvements are discussed.

Reducing the MAC size

A considerable improvement to the protocol is to shrink the size of the MAC. This would allow to reduce storage requirements of both the BisT and FisT solution. With the file size defined as f and the MAC size as m, the relative memory overhead, defined as $rmoh$, is given in Equation (4.1).

$$\begin{aligned} rmoh = \frac{newfilesize}{originalfilesize} - 1 &= \frac{f + numofblocks * m}{f} - 1 \\ &= \frac{f}{(1\,472 - m)} * \frac{m}{f} = \frac{m}{1\,472 - m}, \end{aligned} \tag{4.1}$$

with

$$numofblocks = \frac{f}{(1\,472 - m)}$$

and

$$0 \leq m < 1\,472.$$

Equation (4.1) indicates that using a 32-bit MAC instead of a 256-bit MAC, reduces the relative memory overhead from 2.22% to 0.27%. Although shrinking the size of the MAC reduces the overhead in area, it must be ensured the security is not affected. In [57], NIST discusses the recommended length of a MAC. They disapprove lengths smaller than 32 bits and discourage lengths that are smaller than 64 bits. The latter would result in a 0.55% overhead which is still four times less than the implemented solution.

Optimal use of the individual MACs in chunks

Both in the BisT and FisT solution, every chunk is sent together with a MAC on the chunk. Although not discussed nor implemented in this chapter, this can be useful. If the validation of the MAC on the complete partial bitstream fails, not every chunk needs to be resent. Simply verifying which chunks contain errors, could result in only having to resend the erroneous or missing chunks.

Inter-chunk parameter adjustment

The CR obtained from textbook run-length encoding comes with a downside. Compressing high-entropy data will have a smaller impact because performing run-length encoding on high-entropy data will increase the data size rather than reduce it. Exhaustively searching for the optimal word and length sizes on partial bitstreams chunks can increase the CR. Although not yet used, the chunk header offers the possibility to adjust the word and length sizes (w and l) of the run-length encoding, between chunks.

To find these optimal word and length sizes for all chunks, every possible word and length size should be tried and compared. This iterative process begins by applying all qualifying word and length combinations in a run-length encoding of the partial bitstream. If the most optimal CR results in a chunk-size below the maximum available payload size[10], this particular branch ends. When the maximum available payload size is exceeded, the non-encoded chunk is halved. The iterative search process restarts on both parts. This exhaustive search is visualised in Figure 4.14, while Figure 4.15 shows the tiny modification in the payload generation.

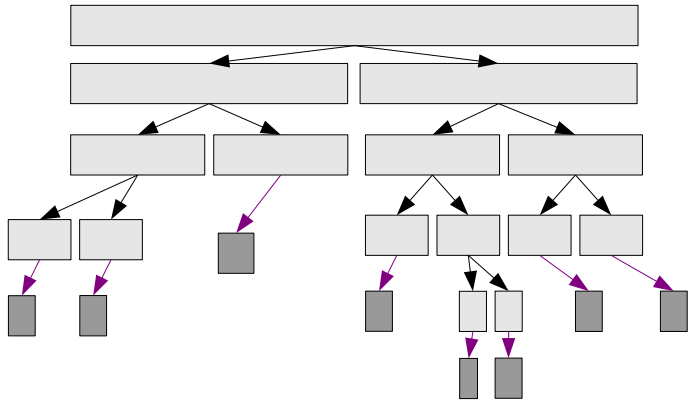

Figure 4.14: Exhaustive search to find optimal word and length sizes

An exploration of the possible gains in CR is executed. The CRs of four encoded bitstreams are compared using different approaches to RLE. Bitstream01 contains a design of a single GPS channel, bitstream02 contains the design of Chapter 3, bitstream03 contains the design of a textbook 1024-bit wide ripple-carry adder, and bitstream 04 contains the design of a video encoder. This

[10]This is the maximum Ethernet payload minus the IP, UDP and reconfiguration headers (1500-20-8-7).

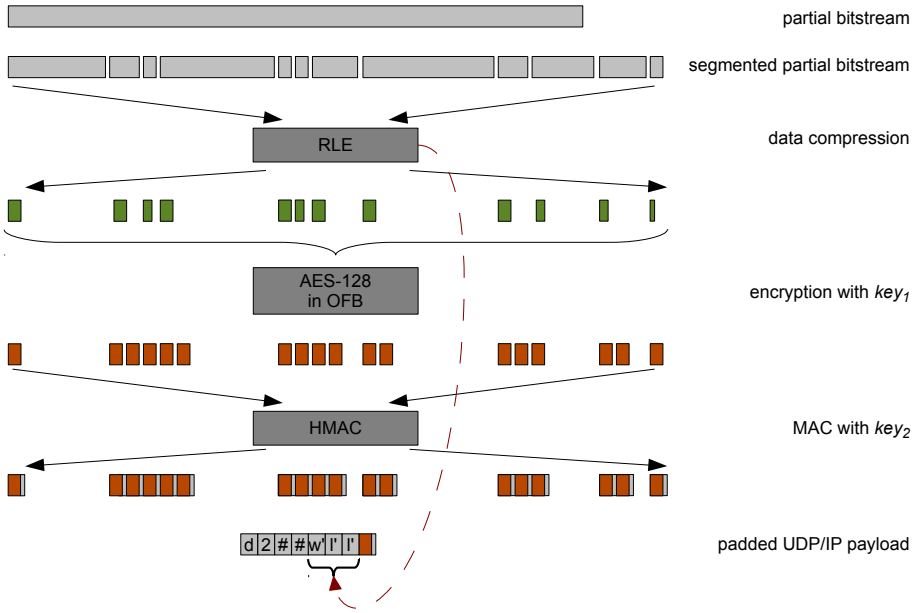

Figure 4.15: The UDP/IP packet payload and its generation procedure used in the FisT solution with inter-chunk parameter adjustment

comparison is shown in Figure 4.16. First there is the compression using RLE of which the worst and best CRs are compared. Subsequently, an exhaustive search on RLE with word and length sizes smaller than 8 bits are added in the comparison. This is indicated with subbyte-run-length encoding (sbRLE). Finaly, the effect on the CR of the inter-chunk parameter adjustment is added to the comparison, with and without subbyte-RLE. To position these results, the CR of compressing the bitstreams with gzip and 7z, two commonly used compression tools, are added.

Change order of MAC and encryption

The order of the MAC and the encryption of the chunks in both the BisT and FisT solution is identical. First, a MAC is calculated on the chunk. Secondly, the concatenation of the chunk and its MAC is encrypted.

In [3], Bellare and Namprempre compare three different approaches in combining an encryption and a MAC. The approach used in this chapter is in their work referred to as MAC-then-encrypt. The two other approaches are:

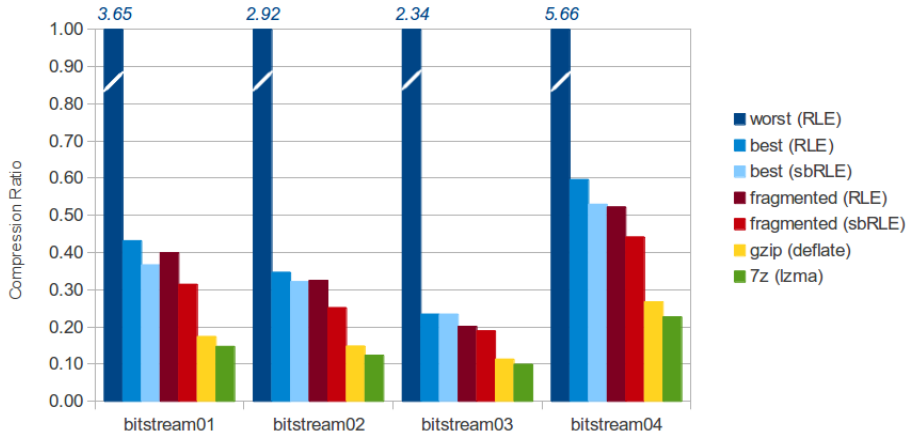

Figure 4.16: Comparison of the CR between textbook RLE, RLE on sub-byte level and RLE with inter-chunk parameter adjustment

Encrypt-and-MAC and Encrypt-then-MAC. Applied to the BisT and FisT solution, the Encrypt-and-MAC solution would come down to concatenating the encrypted chunk and the MAC on the chunk. The Encrypt-then-MAC approach would encrypt the chunk and append a MAC on this encrypted chunk. Bellare and Namprempre make a short comparison of these three methods and their properties. Their result, assuming that the MAC is strongly unforgeable, indicates that switching to Encrypt-then-MAC would offer more complete security. Both for the BisT and FisT solutions, this can be achieved at next to no extra area cost.

Within the European Network of Excellence for Cryptology (ECRYPT), the Directions In Authenticated Ciphers (DIAC) workshop is organised. This workshop has started a competition in search for authenticated encryption algorithms: Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR). The results of this competition could be an alternative for the combination of the MAC and encryption.

4.4 Conclusion

In this chapter a prototype is presented to equip electronic devices, running on an FPGA, with the feature of secure, remote reconfiguration. To obtain this feature a suitable cryptographic protocol is chosen and the required cryptographic primitives to execute the protocol, are implemented. In both the BisT and FisT

solutions, the focus is on area rather than on speed in order to keep the static partition as small as possible, thus leaving maximum resources for the main application.

The full STS protocol ensures data confidentiality, data authenticity and mutual entity authentication between the distributor, who initiates an update or upgrade remotely, and the FPGA in the electronic device. With these features ensured, partial bitstreams can be sent securely. The implementation resists timing attacks and simple power analysis attacks. However, fault injection attacks and DOS attacks are not covered.

Two solutions have been presented. The first solution trusts the PCB on which the FPGA resides, while the second solution only trusts the FPGA. The small differences in the reconfiguration protocol and the implementation are discussed.

This chapter has been published in the proceedings of the International Conference on ReConFigurable Computing and FPGAs [74] and the proceedings of the International Workshop on Reconfigurable Communication-centric Systems-on-Chip [6], and is accepted for publication in the ACM Transactions on Reconfigurable Technology and Systems journal.

Table 4.3: Timing of different types of FPGA reconfiguration

type of reconfiguration	communication channel	size of bitstream [bytes]	time [s] key negotiation	time [s] data transfer	time [s] reconfig-uration	total time [s]
full	JTAG	3 378 281	na	na	na	13.59
partial	JTAG	62 357	na	na	na	2.67
secure, remote, dynamic	network	64 480	3.52	3.82	0.31	7.65
			na: not available			

Table 4.4: Comparison of BisT and FisT solutions with related work targeted at general FPGA reconfiguration

feature	Castillo et al. [8]	Drimer et al. [20]	Kepa et al. [32]	Devic et al. [17]	BisT solution	FisT solution
f1	DES	tbd	AES	**AES-CBC**	**AES-OFB**	
f2	MD5	tbd	nc	**HMAC-256**	**HMAC-256**	
f3	na	na	na	na	**yes**	
f4	na	na	na	na	**no**	
f5	**local TFTP**	tbd	SDRAM	**Ethernet**	**Ethernet**	
f6	na	**yes**	na	**yes**	**yes**	
f7	na	na	na	na	**measures against power**	
f8	**partial**	full	**partial**	full	**partial**	
f9	smfc	external	SDRAM	external/BRAM	BPI Flash	BRAM
f10	nc	(un-)**encrypted**	**encrypted**	yes/nc	**encrypted**	**no**

na: not available
nc: not clearly mentioned
tbd: to be decided at implementation
smfc: Smart Media Flash card

TA: timing analysis
SPA: simple power analysis

f1: encryption
f2: message authentication
f3: entity authentication
f4: trusted third party
f5: communication channel
f6: spoofing & replay proof
f7: SCA measures
f8: reconfiguration
f9: bitstream storage
f10: storage of partial bitstream

Chapter 5

Application: Licensing scheme

This chapter describes the third application of the thesis: a proof-of-concept implementation of a licensing scheme. The original pay-per-use licensing scheme is developed by Maes et al. [41]. They describe the complete scheme but also mention some challenges in building an implementation. Our contribution, which is described from Section 5.2 onward, is twofold: it provides solutions for these challenges and it provides a proof-of-concept implementation of the licensing scheme, by using the GoAhead tool and the differential bitstream generation technique.

5.1 Introduction

5.1.1 Situating the application

Hardware designs are made for a wide variety of applications. The devices for which these designs are tailored are also getting more complex and more powerful. Additionally, the techniques available for developing hardware designs are continuously improved. This makes the design of efficient hardware an increasingly complex task. Rather than starting each hardware design from scratch, reusing already designed components becomes common practice.

With the complexity of individual components increasing, a new market of designing and selling IP cores has been introduced. With reconfigurable

hardware, such as FPGAs, these IP cores can be implemented easily and dynamically. Moreover, their reconfigurability allows IP cores to be software and/or hardware in contrast to the firmware updates of microprocessors which only target software updates.

With the risk of loosing IP, and therefore money, a number of solutions have been proposed. Simpson and Schaumont describe an offline authentication scheme for embedded software IP modules in FPGAs [69]. Besides this software-oriented approach, a number of hardware solutions were proposed [19, 22, 25, 41]. In [22], a proof-of-concept implementation is presented to reconfigure the majority of the FPGA such that it has a design, containing a specific IP core. However, this does not provide a flexible way of obtaining and implementing one or more IP cores. None of the other solutions ([19, 25, 41]) were implemented in practice, but the work of Maes et al. [41] elaborates the most on practical issues. Moreover, their work offers the pay-per-use feature where the system developer pays a price for the IP core per device in which it is instantiated. The scheme of Maes et al. is based on the property of FPGAs to support partial and dynamic reconfiguration.

Although Maes et al. worked out their pay-per-use licensing scheme to a detailed level, it still leaves considerable practical aspects untouched. In this chapter, the practical feasibility of their scheme is evaluated. As already partly indicated by the authors of [41], the proposed architecture is vulnerable to side-channel attacks. Additionally, commercially available tools do not allow nested and flexible placement of IP cores. An improved scheme and architecture are presented that give a solution to the nesting and flexibility issues. In addition, a novel technique is described to decrease the area overhead. In order to achieve a more secure, practically implementable, and smaller solution, the tool flow is based on the academic tool GoAhead [2]. The result is a working FPGA implementation with a small overhead in area. This is the first implementation of a pay-per-use licensing scheme for hardware IP cores.

5.1.2 Threat model

There are four entities that participate in the scheme: the FPGA vendor, the metering authority, the IP core vendor and the system developer. The interactions between the different entities are shown in Figure 5.1. The scheme consists of an initialisation phase and a design phase. Both are detailly described in Section 5.1.3.

Although it is not obvious to achieve in practice, the initialisation is assumed to occur securely. Sending an FPGA between entities will probably result in different people having physical access to the FPGA. Securing this may better

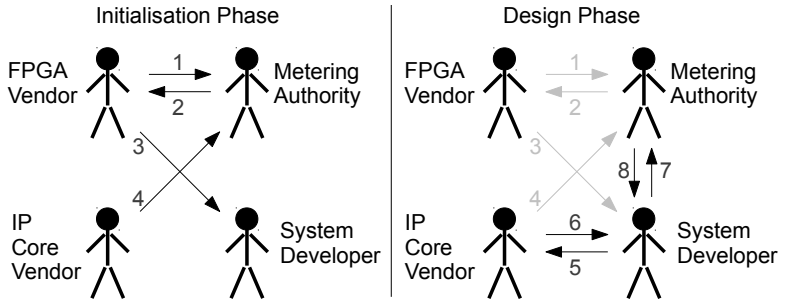

Figure 5.1: The four entities in the scheme and their interactions as described in [41]

be solved in the physical world. It is also assumed that IP specific keys are sent over a secure channel, which might imply other cryptographic techniques.

In the design phase, the communications between the IP core vendor and the system developer require cryptographic protection; they fall outside of the scope of this chapter. The threats, against which this implementation tries to secure, are attacks on the communication between the trusted third party and the system developer.

It is assumed that an attacker can store every message that is sent between the metering authority and the system designer, and that he/she also has physical access to the FPGA. This could enable an attacker to eavesdrop communicated and off-line bitstreams and that he/she can perform power side-channel analysis attacks.

5.1.3 A summary of the scheme [41]

As mentioned above, the scheme consists of an initialisation phase and a design phase that are both described below in detail.

Initialisation phase

When the FPGA vendor produces an FPGA F_i^*, the device can be sent to a metering authority for registration (transaction 1 in Figure 5.1). The metering authority generates a random device key k_i^F and metering key k_i^M and stores these keys in a database together with the ID of the FPGA: ID(F_i). Further, the metering authority stores the device key k_i^F in the secure NVM of the FPGA. This memory is secure in the sense that the stored key can only be

accessed from the bitstream decryption engine. Once the FPGA is configured, the key cannot be read from the reconfigurable fabric. The metering authority will additionally encrypt a bitstream, using the key k_i^F. This bitstream is called the metering bitstream and it contains the metering design M which includes a register that stores the metering key k_i^M. The metering bitstream $B_i(M||k_i^M)$ is computed as $B_i(M||k_i^M) = E_{k_i^F}[b(M||k_i^M)]$, where $E_x[M]$ stands for a symmetric-key encryption of a message M using key x, and $b(x||y)$ stands for a plain text bitstream that implements hardware components x and y. The uninitialised FPGA F_i^* is now transformed into a registered FPGA F_i, which is handed back to the FPGA vendor, together with $B_i(M||k_i^M)$ (transaction 2 in Figure 5.1).

When a system developer buys an FPGA, possibly as component on a development board, which is enabled to use the licensing scheme, the FPGA F_i is delivered, together with $B_i(M||k_i^M)$ (transaction 3 in Figure 5.1).

IP core vendors also need to register their cores through the metering authority. They have to register every offered IP core by providing the metering authority with an ID of the IP core, ID(IP_j), together with a key, k_j^{IP} (transaction 4 in Figure 5.1). Both the metering authority and the IP core vendor store ID(IP_j) and k_j^{IP} in a database.

Design phase

When a system developer wants to obtain and use an IP core in a specific FPGA, the following interactions occur. The system developer requests the IP core identified by ID(IP_j) from the IP core vendor (transaction 5 in Figure 5.1). This results in the IP core vendor sending the bitstream of the IP core to the system developer, encrypted with the key k_j^{IP}, i.e. $B(IP_j) = E_{k_j^{IP}}[b(IP_j)]$ (transaction 6 in Figure 5.1). After transactions 5 and 6 the system developer cannot use the IP core because the decryption key is not yet available.

When the system developer wants to integrate the core in his/her design, the IDs of the FPGA (ID(F_i)) and the IP core (ID(IP_j)) are sent to the metering authority (transaction 7 in Figure 5.1). The metering authority generates a license $K_{i,j}^{IP}$, with $K_{i,j}^{IP} = E_{k_i^M}[k_j^{IP}]$, which is sent to the system developer (transaction 8 in Figure 5.1).

The system developer then configures the FPGA F_i with bitstream $B_i(M||k_i^M)$, which allows the key k_i^M to get on the FPGA. Using k_i^M, the system decrypts the license ($K_{i,j}^{IP}$) to obtain k_j^{IP}, which is used to decrypt the encrypted IP core $B(IP_j)$ on the FPGA, in order to configure the IP core in the system developer's design through partial reconfiguration.

During the design phase, after all transactions in Figure 5.1 have been performed, the architecture implemented on the FPGA is altered a few times. This evolution is depicted in Figure 5.2, where the white area represents the reconfigurable resources of the FPGA, while the grey area holds additional dedicated components available on the die of the FPGA. It is pointed out that the ICAP is permanently present in the reconfigurable part. In every step of Figure 5.2 incoming data is used to update registers or to reconfigure the FPGA. The targeted components of the incoming data are indicated by the black, dashed arrows.

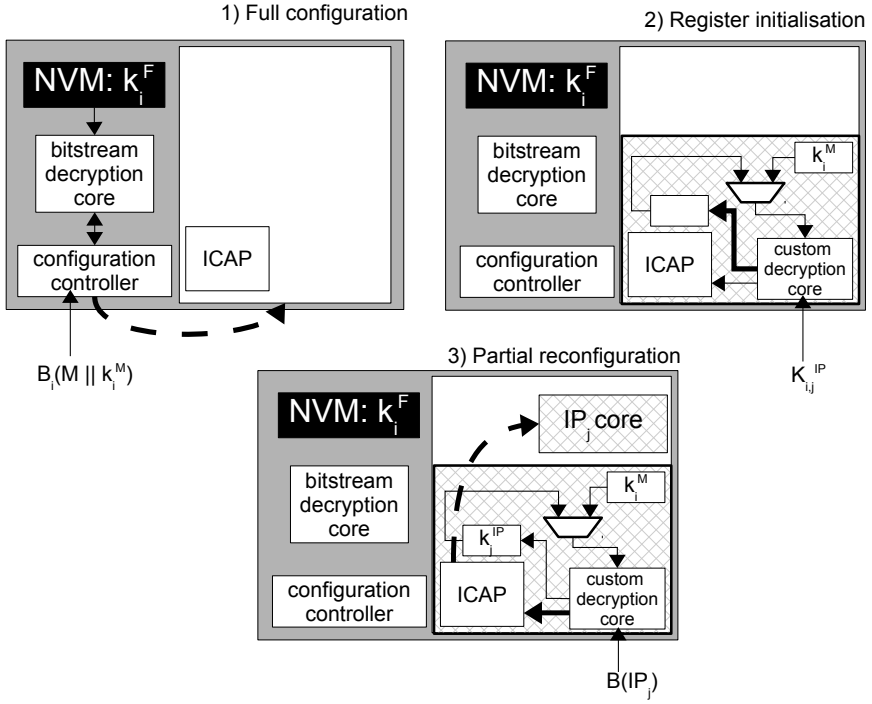

Figure 5.2: The evolution of the FPGA architecture in [41] during the design phase. The Non-Volatile Memory (NVM) stores the device key for the on-chip decryption core.

The top left image visualises how $B_i(M||k_i^M)$ configures the metering design and the metering key in the FPGA through a full configuration. This configuration uses the on-chip bitstream decryption core which is available on the FPGA, using the key k_i^F which is stored in NVM. The partial reconfigurations of the FPGA that follow in a later phase do not alter the metering design. Therefore, we refer to the part of the FPGA that holds the metering design as the 'static part',

while the remainder of the FPGA is reserved for the system developer's design together with the IP cores.

The top right image in Figure 5.2 shows the second step, in which the metering design, consisting of a custom AES decryption core and two registers (of which one contains the metering key k_i^M), is already present in the static part of the FPGA. The license $K_{i,j}^{IP}$ is decrypted by the custom decryption core on the reconfigurable fabric of the FPGA, using key k_i^M, which results in the initialisation of the already implemented register for k_j^{IP}. It should be noted that this step performs no configuration, but the initialisation of a key in a register.

The bottom image shows the incoming, encrypted bitstream $B(IP_j)$, containing the obtained IP core. This bitstream is decrypted by the custom decryption core, using key k_j^{IP}, and is routed to the ICAP. This results in a partial reconfiguration of the FPGA to implement the design of the IP core.

5.2 Implementation

5.2.1 Issues in implementing the licensing scheme

As already partly indicated by Maes et al., the implementation of the licensing scheme leads to a number of practical issues. In this section we explain the two most important issues.

Side-channel security of the embedded AES core

The original scheme uses the dedicated AES bitstream decryption core on the die of the FPGA for the full configuration of the initial system. In [41], Maes et al. already mention the work of Moradi et al., that presents a side-channel attack on the built-in AES core [51]. The attack reveals the key that is used to decrypt encrypted bitstreams. In the licensing scheme, this means that k_i^F can be revealed during the decryption of $B_i(M||k_i^M)$. With k_i^F, the attacker can decrypt $B_i(M||k_i^M)$ outside the FPGA to obtain k_i^M. This would reveal k_j^{IP} from the license, which can finally be used to decrypt the partial bitstream containing the obtained IP core. An in-depth study of this bitstream can reveal implementation details of the IP core which threatens the work of the IP core vendor. Note that Maes et al. use a custom AES decryption core for the partial bitstreams, since decryption of a partial bitstream was not supported by the on-chip core.

Nested and flexible integration of IP cores

As shown in Figure 5.2, the metering design resides in the static part of the FPGA. The system developer's design is a partial module that covers the remaining part of the FPGA. Since the obtained IP core will be placed inside the system designer's hardware, the IP core needs to be nested as a partial module inside the system developer's design. This requires the ability of performing nested partial reconfiguration, which is not supported by commercial tools. Moreover, flexible placement of IP cores is not allowed either, which means that an IP core can only be implemented at a predetermined location on the FPGA. Maes et al. support this by having the IP core vendor tailor the obtained IP core to the system developer's needs. However, this increases the design time and the IP core price and results in a non-flexible, non-scalable solution.

5.2.2 Overcoming the issues

Side-channel security of the embedded AES core

The original architecture in [41] uses the on-chip AES decryption core for full configuration and uses a custom AES core in the reconfigurable logic of the FPGA in order to decrypt licenses and partial bitstreams, as explained in Section 5.1.3. To solve the side-channel security issues of the on-chip AES core, there are two solutions. Either the on-chip AES core needs to be replaced by a side-channel secure core or a work-around needs to be found based on the existing FPGA technology. Because the former is the responsibility of the FPGA vendor and because a solution for existing FPGAs is desirable, the vulnerable on-chip AES core is not used. Instead only the custom AES core is used in the reconfigurable logic for all decryptions.

This core needs to be configured in the FPGA which is done with an encrypted bitstream, B_{init}, that also contains a connection to the ICAP and a storage unit for both the metering key k_i^M and the IP core key k_j^{IP}. Next, the encrypted bitstream $B_i(M||k_i^M)$ is sent to the FPGA and gets decrypted by the custom AES core. According to the original scheme, $B_i(M||k_i^M)$ contains a register that holds k_i^M, an AES decryption component, and a connection to the ICAP. Since the AES decryption component and the connection to the ICAP were already implemented by B_{init}, the only new component in $B_i(M||k_i^M)$ is the metering key k_i^M. The encrypted bitstream $B_i(M||k_i^M)$ is decrypted with key k_i^F, which is available inside the FPGA after the initialisation phase. The implementation of the custom AES core is based on the work of Moradi et al. [52].

Note that no precautions have been taken to prevent an attacker from altering the encrypted bitstream B_{init} through the techniques of Moradi et al. [51]. This full reconfiguration uses the vulnerable on-chip core, but the design of B_{init} does not contain sensitive data. However, an attacker could make connections between the key storage and the outside world to simply eavesdrop every key. Because doing this on a placed-and-routed design is considered to be difficult, we do not take precautions to prevent this attack in this proof-of-concept implementation.

Nested and flexible integration of IP cores

Commercially available tools for partial reconfiguration do not allow partial modules to be nested. Koch et al. developed an academic tool called ReCoBus-Builder [37] that evolved into the GoAhead tool [2], which does allow nested partial reconfiguration. This tool heavily relies on the Xilinx Design Language (XDL) [81] to achieve its unique features. Therefore, the use of the tool binds the scheme to Xilinx FPGAs. Further, GoAhead also allows the flexible placement of reconfigurable modules in contrast to a predetermined location of modules using commercial tools. This makes our solution more practical, cheaper and less cumbersome for both the IP core vendor and the system developer. An additional benefit of using GoAhead is that partial reconfiguration on the Spartan-6 FPGA family becomes possible, which is not feasible using commercial tools.

5.2.3 Additional improvements to the architecture

In order to decrease the resource occupation of the static partition that handles the licensing scheme, an alternative method for key storage is implemented. In the original scheme, only the metering key (k_i^M) and the IP core key (k_j^{IP}) are stored in the flip-flops of the FPGA, since these are the keys used for the custom decryption core. The device key (k_i^F) is stored in a memory element that drives the built-in AES decryptor. As explained in Section 5.2.2 our solution uses the custom AES core for all decryptions, which means three keys have to be stored. The next section describes how traditional storage of the keys is done, while the section thereafter explains how the novel approach allows the storage of three keys using only half the area compared to the storage of two keys using the traditional method. Moreover, in the traditional setting, two additional reconfigurable partitions are needed for the storage of the keys, while our approach does not need any further partitioning in the static partition. This leads to additional savings in area and timing, since the interconnection of partial modules with the rest of the FPGA introduces an overhead in area and timing.

The traditional way: key storage in slice flip-flops

Upon initial configuration, the registers for storing k_i^M and k_j^{IP} are empty. During two partial reconfiguration steps the partial bitstream $B_i(M||k_i^M)$ and the license $K_{i,j}^{IP}$ store k_i^M and k_j^{IP} in these registers. Assuming AES-128 is used for encryption, each register holds a 128-bit key. Given that one slice contains eight flip-flops (for a Xilinx Spartan-6 FPGA), the two registers require 32 slices.

The improved way: key storage in configuration memory

In order to reduce the overhead used by the key registers, the storage of the keys is moved from the slice flip-flops to the 'configuration memory' of the FPGA by using the configuration bits of the LUTs.

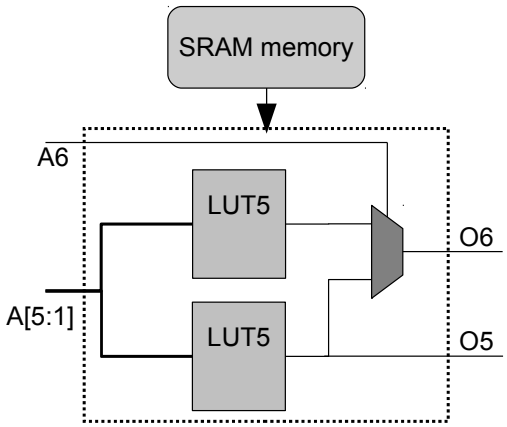

Figure 5.3: Architecture of a LUT6 in a Spartan-6 FPGA

Figure 5.3 shows the architecture of a LUT6 of a Spartan-6 FPGA with 6 inputs and 2 outputs. This LUT6 can be configured as a single 6-to-1 LUT or as two 5-to-1 LUTs. Any function with n inputs (with $1 \leq n < 6$)can be configured in a LUT, resulting in 2^n possible functions. Among these functions are two functions that map any given input to '0' or to '1', e.g. F1 and F16 in Table 1.2. Configuring both 5-to-1 LUTs through the SRAM memory to one of these two functions, turns a single LUT6 in a 2-bit ROM. The value of the ROM is stored in the configuration memory. Since we need to provide the AES core with 128 key bits in parallel, we need 64 LUTs or 16 slices for the storage of one key.

Due to reasons explained above, there are three keys to be stored, namely k_i^F, k_i^M and k_j^{IP}. By altering the truth table of the LUT-as-2-bit-ROM as shown in Table 5.1, the single LUT can hold 2 bits of all three keys. This means that the 16 occupied slices can store all three keys. From a functional point of view, the achieved behaviour could be represented as shown in Figure 5.4. The configuration bits of the LUT determine the value of the 2-bit ROMs and therefore determine 2 bits of each key. Each LUT has a truth table as shown in Table 5.1.

Table 5.1: Truth table of the LUTs in the key storage unit

A1	A2	A3	A4	A5	A6	O5	O6
0	X	X	X	X	X	$k_i^F[n]$	$k_i^F[n+1]$
1	0	X	X	X	X	$k_i^M[n]$	$k_i^M[n+1]$
1	1	X	X	X	X	$k_i^{IP}[n]$	$k_i^{IP}[n+1]$

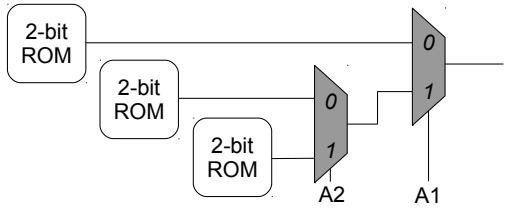

Figure 5.4: The LUT configuration in the key storage unit

The group of 16 slices which stores the three keys is referred to as the 'key storage unit'. To prevent the design tool from optimising this construction away, the key storage unit is implemented as a hard macro [80]. When instantiating the key storage unit, all key bits are initialised to '0'. An update of the keys is achieved through the reconfiguration of the 16 slices of the key storage unit.

In order to be able to switch between the three keys for the AES decryption core, the selection inputs $A1$ and $A2$ of the multiplexers in Figure 5.4 have to be altered. To be able to switch between keys through reconfiguration, the same approach of using a LUT as a 2-bit ROM can be used again. The truth table of this LUT is shown in Table 5.2. The slice in which this LUT resides is referred to as the 'key switching unit'. Upon initialisation, $A1$ and $A2$ are forced to '0' because the first key that needs to be offered to the AES core is k_i^F.

This novel technique of storing the different keys, alters the protocol of changing between keys. In the original licensing scheme of Maes et al. [41], receiving a

Table 5.2: Truth table of the LUT in the key switching unit

A1	A2	A3	A4	A5	A6	O5	O6
X	X	X	X	X	X	A1	A2

new key happens in a single iteration. This is possible because both k_i^M and k_j^{IP} are stored in two separate registers. Using the proposed technique requires two steps for storing a new key: 1) storing the key in the key storage unit and 2) updating the selection bits in the key switching unit.

In the first step, the incoming partial bitstream (containing the new key k_{new}) is decrypted with the currently used key k_{active} and forwarded to the ICAP to update the key storage unit. This has no effect on the value of k_{active}, used for decrypting the incoming bitstream, because the key switching unit is not yet updated.

It is only upon receiving an update of the key switching unit that k_{new} gets used in the AES core. Naively using the same method as updating the key storage unit would result in switching the key **during** decryption of the incoming partial bitstream. To prevent this, the partial bitstream to update the key switching unit has to be received, decrypted and temporarily stored **before** routing it to the ICAP. Achieving this can be done by adding a FIFO that stores the decrypted partial bitstream. Because the partial bitstream for updating the key switching unit (1 slice) is small, the FIFO is small as well.

5.2.4 Novel architecture and tool flow

Novel architecture

The architecture residing in the static partition, handling the licensing scheme, is depicted in Figure 5.5. The key switching unit sends the selection signals $A1$ and $A2$ to the key storage unit in order to determine which of the three keys (k_i^F or k_i^M or k_j^{IP}) is used in the AES core. The output of the custom AES core is forwarded through a bit swapper to the ICAP. The bit swapper makes sure the bits of the decrypted bitstreams are routed in the correct order to the ICAP. The need for the FIFO is explained in Section 5.2.3. In our proof-of-concept implementation, encrypted bitstreams are sent to the AES core over a UART [62]. In an industrial implementation the communication interface needs to be replaced by an interface with a higher throughput that is accessible through the Internet.

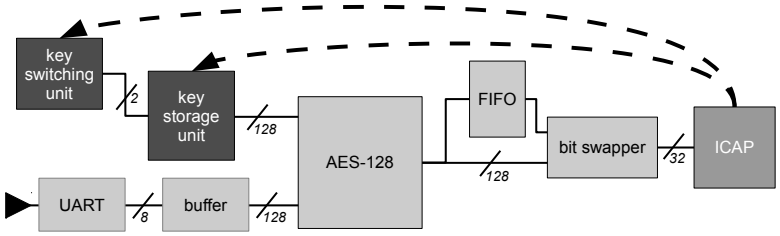

Figure 5.5: The architecture residing in the static partition, handling the improved licensing scheme

Figure 5.6 shows a floorplan for the nested partial reconfiguration of the FPGA. The largest block (Static) consists of all reconfigurable resources on the FPGA. The two rightmost components are the key switching unit (KSwU) and the key storage unit (KStU). These units reside in the static partition. The darker coloured component on the left is a reconfigurable partition (Design), which holds the design of the system developer. Up until this point there is a single static partition with a single reconfigurable partition. The IP core which is acquired for the design of the system developer is a reconfigurable partition as well, encapsulated in another reconfigurable partition (Design). The difference compared with the others is that this partition does not reside in the static partition, but in a reconfigurable partition.

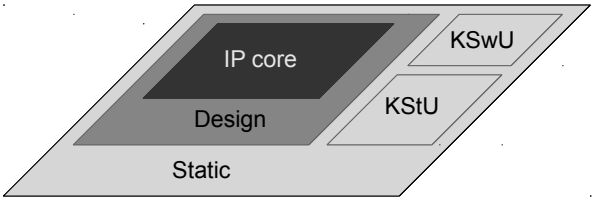

Figure 5.6: Graphical representation of the nesting levels, where the Static partition occupies one level; the Design partition, the key storage unit (KStU) and key switching unit (KSwU) form the first level and the IP core(s) form the second nesting level.

The evolution of the FPGA architecture in the design phase of the novel architecture is shown in Figure 5.7.

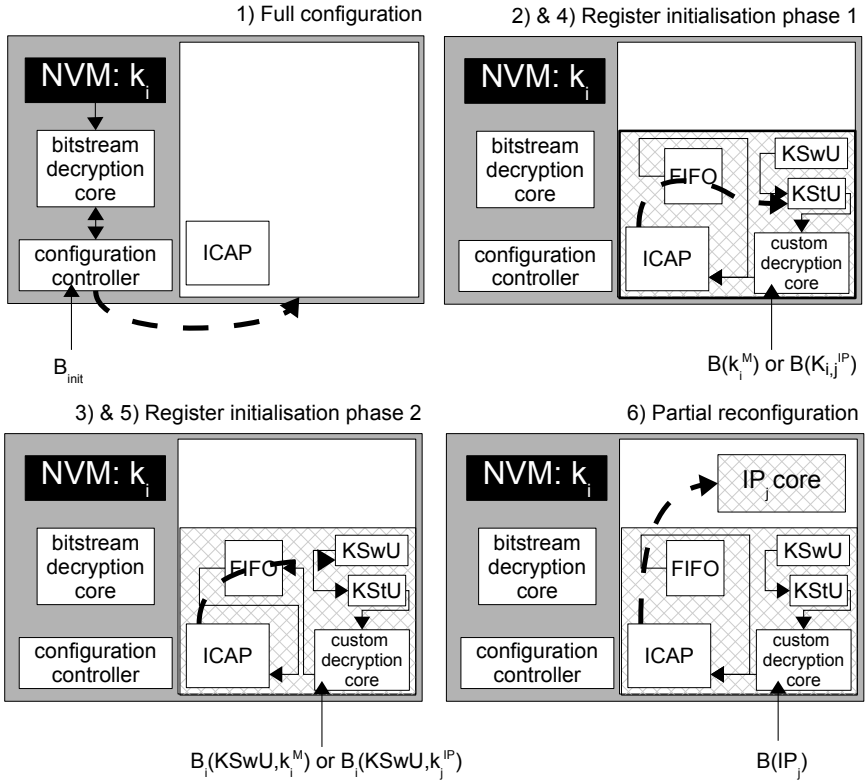

Figure 5.7: The evolution of the novel FPGA architecture during the design phase. The NVM stores the device key for the on-chip decryption core.

Novel tool flow

Xilinx offers the PlanAhead tool for partial reconfiguration. This tool generates a partial bitstream for every possible configuration of the static and reconfigurable partitions. For example, a design with two reconfigurable partitions for which the first reconfigurable partition has two designs and the second reconfigurable partition has three designs, ends up with six full bitstreams and six times two partial bitstreams. Moreover, it is pointed out that nested partial reconfiguration is not possible with the PlanAhead tool.

To achieve a proof-of-concept implementation for the licensing scheme, the partial bitstreams are generated with the differential bitstream technique. To enable the nesting of partial modules, we use the GoAhead tool [2].

In order to reconfigure the key storage unit and the key switching unit through partial reconfiguration, these partial bitstreams are generated by making differential bitstreams with respect to b_{init} and to $b(M_i||k_{iM})$, for k_i^M and k_j^{IP} respectively. To use this technique, first a full bitstream is generated. This has uninitialised hard macros for the key storage and key switching units. Then, a second full bitstream is generated, which contains the desired modification to one of the units. To generate a differential bitstream which only contains the modifications in the second full bitstream, the final 'bitgen' step has to be rerun, as depicted in Figure 5.8 . Additional parameters[11] are applied in the bitgen step, to make a differential bitstream and indicate the full bitstream which serves as a reference. With this option, the resulting bitstream only contains the frames in which the second full bitstream differs from the first full bitstream. For this reason it is important that no other differences between these full bitstreams exist.

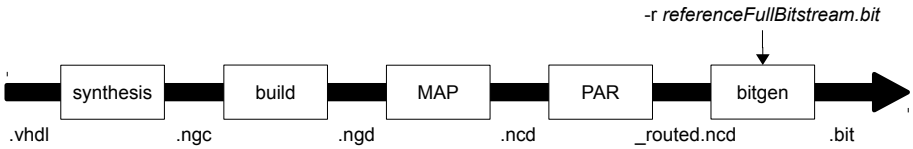

Figure 5.8: The processing steps to obtain a differential bitstream, with the corresponding file extensions of the bitstream generation process

The full bitstream, b_{init}, is restricted in the area which can be used: only the 'Static partition' from Figure 5.6 is available. To achieve this through GoAhead, the tool applies restrictions on the placement and routing tools to exclude the usage of the primitives in a certain area. To prevent the placer tool from using primitives from the restricted area, an additional 'user constraint file' is added. To prevent the routing tool from using connections in the restricted area, a 'StaticBlocker' is used. This blocker is a component that occupies every restricted net. This blocker is merged with the result of the placer tool. This '_blocked.ncd' design is then routed by the Xilinx router after which the 'StaticBlocker' is removed. The modified chain of processing steps is shown in Figure 5.9.

The generation of a partial bitstream for the 'design', shown in Figure 5.6, is done by combining the GoAhead approach with the differential bitstream generation technique. First, a full bitstream is generated that contains the design. This generation is analogous to the generation of the full bitstream of the static partition, with a difference in the additional user constraint file and a substitution of the StaticBlocker by a PartialBlocker. These differences

[11] In the Xilinx design suite version 14.5, this parameter is '-r referenceFullBitstream.bit'.

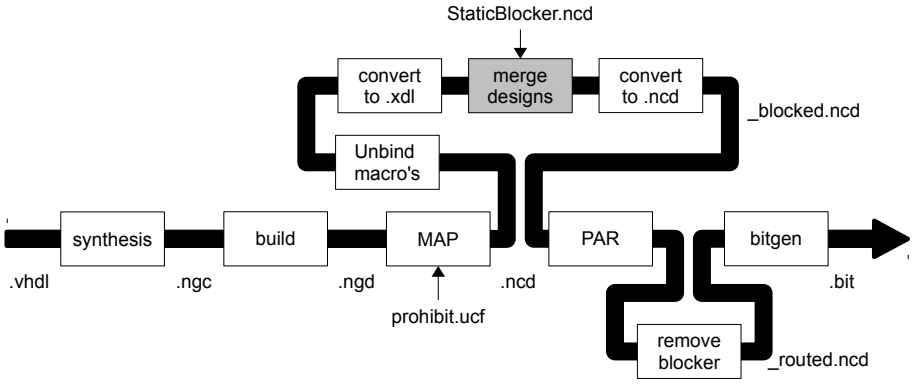

Figure 5.9: The processing steps through GoAhead to obtain a full bitstream, with the corresponding file extensions of the bitstream generation process

prevent the placer and routing tools to use resources in the static partition. Secondly, a differential bitstream is generated between the full bitstream which only contains the 'design' and an empty bitstream. This results in a partial bitstream which only reconfigures the 'design' partition.

The techniques to generate the static and partial bitstreams as explained above, can be extended to an additional nesting level. For more details and a more in-depth explanation about generating the full and partial bitstreams using GoAhead, we refer to the GoAhead documentation [2].

5.2.5 Implementation results

Area

The architecture has been implemented on a Digilent Atlys board, containing a Xilinx XC6S-LX45 FPGA. The FPGA resources occupied for the execution of the licensing scheme are shown in Table 5.3. The table illustrates that the majority of resources is dedicated to the AES core. Since the system developer needs as much free space as possible for the implementation of his/her own design in combination with licensed IP cores, the number of remaining resources available on the FPGA is important. Therefore, the table shows the total number of relevant FPGA resources we used for our proof-of-concept implementation, but also for the smallest and the largest member of the Spartan-6 FPGA family.

Although, Table 5.3 shows that the overhead is relatively small, it is still too large to fit the smallest FPGA family member.

Table 5.3: Occupied resources by the licensing scheme for different Spartan-6 FPGAs

	Occupied slices	16K BRAM	DSP 48A1	min clk period [ns]	num of clock cycles[1]	duration [ns]
AES	646	0	0	5.00	330	1 650
static	841	4	0			

[1]: number of clock cycles for a single encryption

	Available slices	BRAM	DSP48A1
XC6SLX4	600	12	8
XC6SLX45	6 822	116	58
XC6SLX150	184 304	268	180

Speed

A full bitstream for the XC6SLX45 has a size of 1 484 514 bytes. The partial bitstream to update a key in the key storage unit, including padding for communication and encryption, is 2 232 bytes. The partial bitstream to alter the output of the key switching unit is 1 432 bytes. In our proof-of-concept implementation, a UART interface with a baud rate of 115 200 Bd is used. This results in a duration of 155 ms and 99 ms for the communication of the two partial bitstreams, respectively. For an update of the key in the key storage unit, 140 AES decryptions have to be performed, which takes 0.462 ms (at a speed of 3.3 μs per decryption, as reported in Table 2.13 with a system clock of 100 MHz). For an update of the key switching unit, 90 AES decryptions have to be performed, which takes 0.297 ms. In total a key update takes 155.462 ms (155 ms + 0.462 ms) and a key switch takes 99.297 ms (99 ms + 0.297 ms). In this proof-of-concept implementation, the communication speed forms the bottleneck. It is clear that a communication channel with a higher bandwidth is necessary for the practical deployment of the system.

5.3 Results

Although the implementation presented in this chapter is the first working implementation of a pay-per-use licensing scheme, a number of issues are still

open for improvement; the most relevant ones are mentioned.

The hardest challenge is the storage of the FPGA key k_i^F. Storing this key in a secure and non-volatile way is tough to accomplish. In the solution described above it is assumed the FPGA has non-volatile memory but that is not a common nor an inexpensive feature. A solution for this problem might be obtained from Physical Unclonable Functions (PUFs) [23].

In a proof-of-concept implementation, the slow serial interface used in this chapter can be used. However, this is not acceptable in an industrial setting. A communication channel with a higher bandwidth and Internet connectivity should be included to make the solution usable in a real-life setting. The solution from Chapter 4 can be used, but its relatively large overhead would affect the available reconfigurable resources on this smaller device even worse.

5.4 Conclusion

This chapter describes a practical evaluation of the licensing scheme presented by Maes et al. in [41]. Our contribution is threefold: a number of feasibility and usability issues in the original scheme are tackled, additional improvements that decrease the area overhead are implemented and a proof-of-concept implementation of the licensing scheme is obtained.

The novelty in decreasing the area overhead consists of moving the key storage from the slice flip-flops to the configuration memory. To obtain the proof-of-concept implementation we used the GoAhead tool in combination with the differential bitstream generation technique.

Even if this chapter resulted in a proof-of-concept implementation, future work could improve on this. A side-channel robust implementation of AES with an authenticated mode of operation is a first possibility. A second is to incorporate a communication channel with a higher throughput.

This results from this chapter have been submitted to the Journal of Cryptographic Engineering.

Chapter 6

Conclusions and Future work

6.1 Conclusions

This thesis explores options for the implementation of applications containing security challenges on FPGAs.

The first part of our work consists of the implementation of four cryptographic primitives on an FPGA. In this part, our most important contribution is the implementation of an elliptic curve processor (ECP) over prime fields based on a novel architecture. This resulted in the smallest FPGA implementation at the time of publication. Another contribution is our design approach, based on the wrapper component to interface the hardware primitives. This unified interface makes replacing certain primitives by newer versions or even other primitives seamless, as is illustrated with multiple implementation variants of the ECP and AES-128.

In the second part of our work, we use the implemented primitives in three specific applications.

The first application uses hardware to strengthen a distributed logging scheme. The added value of the implementation of certain components in hardware is the reduction of the level of trust that has to be put in the log server. Our work shows that it is feasible to move the implementation from software to hardware. We also summarise how the hardware architecture can be altered in order to outperform the software reference implementation.

The second presented application shows how a reconfigurable chip in an electronic device can be updated to prolong the device's lifetime. We introduce two architectures, one based on the BisT solution (Board is Trusted) and one based on the FisT solution (FPGA is Trusted). Next to the minimised occupation of FPGA resources, our contribution consists of the introduction of additional features such as: entity authentication, countermeasures against side-channel analysis attacks, and the possibility of avoiding a trusted third party, both in the BisT and FisT solutions. In addition, the FisT architecture represents the first single-chip solution for secure remote configuration of FPGAs.

The last presented application handles the practical difficulties of implementing a hardware IP core licensing scheme. Our research starts from a theoretical proposal of a pay-per-use licensing scheme, while our contribution is the improvement of the scheme towards practical feasibility and minimal resource occupation. We do this by using a novel storage technique in combination with an academic design tool. To the best of our knowledge, this implementation is the first practical implementation of a hardware IP core licensing scheme.

The implementations of these three applications results in a number of conclusions. A first conclusion is that the use of FPGAs in end products should be reconsidered. Their reconfigurable nature provides features that are not feasible in ASICs such as: remote device updates, flexible usage of IP cores, and shorten time-to-market. With these additional features, the higher unit cost of FPGAs could be justified. The overhead that comes with these unique features is still substantial, but with the increasing size of the latest FPGAs this overhead can be afforded.

Secondly, making an implementation for an application like the remote reconfiguration of FPGAs and the implementation of the IP licensing scheme, combines a number of different fields of research. When combining different fields of research it is a challenge to determine the depth level of each involved field. These choices heavily depend on the application, both its functionality and its available budget. Moreover, the number of entities, the level of trust that is put in each entity, together with the hierarchical structure of these entities play an important role. On the other hand, the considered threat model also has a substantial influence on the choices of how deep to go in a certain field of research. No matter how these parameters turn out, bringing together these different fields is a challenging endeavour.

A third conclusion is that it is not straightforward to implement one general solution for cryptographic primitives. Different applications require different levels of security robustness in their implementations. This lack of design portability forces us to foresee different implementation variants to meet the requirements of specific applications. The use of generics in the general

implementation of primitives contributes in finding the required balance between occupied resources and throughput.

As a final conclusion, it is clear from this research that not all features, which are technically possible, are supported in today's tools. The application in Chapter 5 illustrates this. Due to convincing economical reasons, FPGA vendors keep certain aspects of their products private, thus forcing developers to use their tools, or at least parts of them. This places the vendors in a situation in which other commercial parties, interested in developing implementation tools, cannot compete with them.

6.2 Future work

An anonymous statement, states: 'A design is, what a designer has, when time and money run out.'. For this work, subsequent research can go more in depth on the topics presented in this thesis, or can treat broader related topics.

6.2.1 Future work in depth

In the context of Chapter 2, further specialising and optimising the ECP to narrow the gap with the current state-of-the-art implementations, is a first direction for follow-up research. This could be done by tailoring the hardware design on some FPGA-specific features or to dedicated elliptic curves.

A second direction for future work, related to Chapter 3, is the speed-up of the implemented secure logging scheme, such that it outperforms the software reference implementation. Also, a design and implementation for the backup system can be realised.

The results of Chapter 4 might be improved by exploring dedicated features of new heterogeneous implementation platforms, such as Xilinx' Zynq platform. Further, doing an in-depth study on the redundancy in an FPGA bitstream in combination with the state-of-the-art in data compression, will result in achieving better compression ratios, improving the FisT solution in Chapter 4. Moreover, to enable the use of the FisT solution for any given FPGA, allowing a reasonable static partition, could be obtained by segmenting the single reconfigurable partition. This reduces the size of the partial bitstream, which lowers the storage requirements in the static partition.

Another direction to follow from the results of Chapter 4 is to transform the cryptographic components and the CPU's instruction memory into

reconfigurable modules. With this improvement of the static system, the cryptographic setting can be updated as well. This would be a desirable feature in devices with FPGAs aiming at a long life-cycle. In such a setting, it might be expected the cryptographic overhead needs updates.

The use of Physically Unclonable Functions (PUFs) is the next step in securing the licensing scheme of Chapter 5. Also Chapter 4 would benefit from this to store the private key.

6.2.2 Future work in breadth

This thesis can serve as a foundation to build larger and/or more industry-ready applications. It could be used as a starting point to completely implement one single application, with all accompanying aspects.

The dynamic and partial reconfiguration techniques used in this thesis can also be used to implement countermeasures for side-channel attacks by randomising the hardware implementations while maintaining their functionality. This was first published by Mentens et al. in [46].

Further, this thesis can trigger research on economics. It would be very interesting to investigate the scale of applications that would economically benefit from having reconfigurable hardware. Quantising the volume of a product together with an estimation of the saved costs coming from the benefits of using reconfigurable hardware, could pinpoint the bottlenecks for the introduction of reconfigurable hardware in marketable products.

Appendix A

Finite state machines

A.1 ECP FSM

Figure A.1 shows the states and their succession of the FSM, used in the ECP.

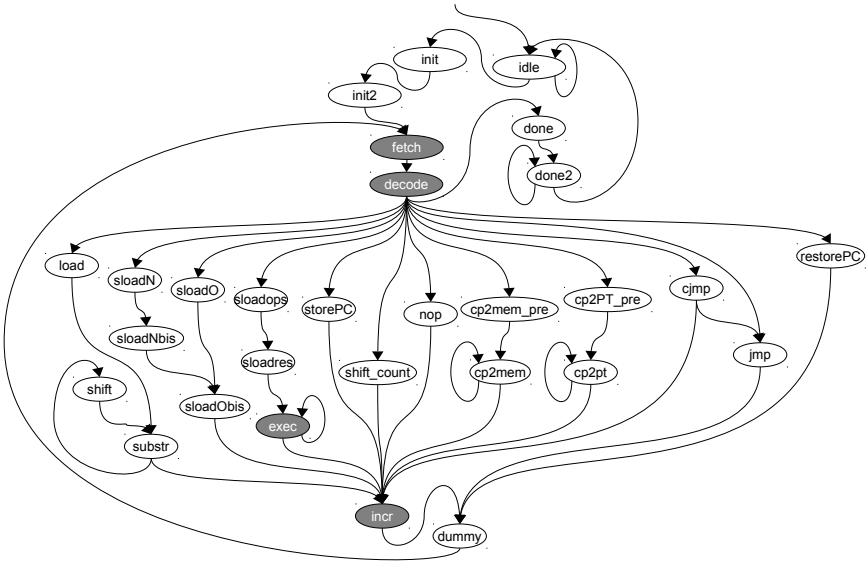

Figure A.1: States of the ECP FSM

A.2 SHAMAC FSM

Figure A.2 shows the states and their succession of the FSM, used in the Hash/HMAC component.

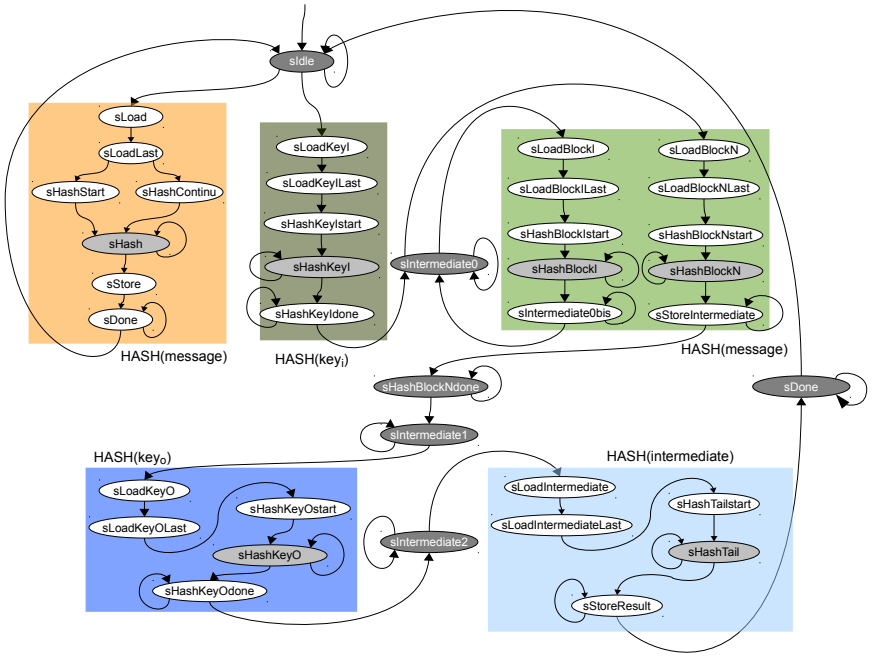

Figure A.2: States of the SHAMAC FSM

Appendix B

ECP source code

```
CP2MEM 0 0
RESTOREPC 0

EXEC PX * R**2 = DIFFX          // P(x,y) -> Q(x,y,z)
EXEC PY * R**2 = DIFFY
EXEC ZM * ZM = T3
EXEC T3 * ZM = T4
EXEC DIFFX * T3 = QX
EXEC DIFFY * T4 = QY
EXEC ZM + ZERO = Q
RESTOREPC 0

EXEC QX * T1 = T4               // qx, qy, T1(=z^-1)
EXEC QY * T1 = T5               //    -> PX, PY (in T6, T7)
EXEC T4 * ONE = T6
EXEC T5 * ONE = T7
RESTOREPC

LOAD 35                         // qz^(p-2) -> T1
EXEC QZ + ZERO = T1
EXEC QZ + ZERO = T2
EXEC T1 * T1 = T1
CJMP 21 0 0
EXEC T1 * T2 = T1
SHIFT
CJMP 18 1
```

```
RESTOREPC 0

EXEC QZ * QZ = T1                    // 2Q -> S
EXEC T1 * AM = T1
EXEC QX * QX = T2
EXEC T2 + T2 = T3
EXEC T2 + T3 = T3
EXEC T1 + T3 = T1
EXEC QY * QZ = T2
EXEC QX * QY = T3
EXEC T3 * T2 = T3
EXEC T1 * T1 = T4
EXEC T3 + T3 = T5
EXEC T5 + T5 = T6
EXEC T6 + T6 = T5
EXEC T4 - T5 = T4
EXEC T6 - T4 = T5
EXEC T1 * T5 = T5
EXEC T2 * T2 = T6
EXEC T6 + T6 = T6
EXEC T6 + T6 = T6
EXEC T6 + T6 = T6
EXEC QY * QY = T7
EXEC T6 * T7 = T7
EXEC T5 - T7 = SY
EXEC T2 * T6 = SZ
EXEC T2 * T4 = T6
EXEC T6 + T6 = SX
RESTOREPC 0

EXEC SX * QZ = T6                    // Q <= S + Q,   S <= 2S
EXEC SX * QX = T1
EXEC SZ * QZ = T4
EXEC QX * SZ = T2
EXEC T6 - T2 = T3
EXEC T3 * T3 = QZ
EXEC DIFFX * QZ = T5
EXEC AM * T4 = T7
EXEC T1 + T7 = T1
EXEC T2 + T6 = T2
EXEC T1 * T2 = T1
EXEC T4 * T4 = T2
```

```
EXEC BM * T2 = T7
EXEC T1 + T7 = T1
EXEC T1 + T1 = T1
EXEC T1 - T5 = T5
EXEC T7 + T5 = T5
EXEC T7 + T5 = QX
EXEC AM * T2 = T2
EXEC T6 * T6 = T1
EXEC T1 + T2 = T1
EXEC T2 + T2 = T2
EXEC T1 - T2 = T2
EXEC T2 * T2 = T2
EXEC T6 * T1 = T1
EXEC T4 * T7 = T7
EXEC T1 + T7 = T1
EXEC T6 * T7 = T7
EXEC T7 + T7 = T7
EXEC T7 + T7 = T7
EXEC T7 + T7 = T7
EXEC T2 - T7 = SX
EXEC T4 * T1 = T6
EXEC T6 + T6 = T6
EXEC T6 + T6 = SZ
RESTOREPC 0

EXEC QX * SZ = T6                    // S <= Q + S,   Q <= 2Q
EXEC QX * SX = T1
EXEC QZ * SZ = T4
EXEC SX * QZ = T2
EXEC T6 - T2 = T3
EXEC T3 * T3 = SZ
EXEC DIFFX * SZ = T5
EXEC AM * T4 = T7
EXEC T1 + T7 = T1
EXEC T2 + T6 = T2
EXEC T1 * T2 = T1
EXEC T4 * T4 = T2
EXEC BM * T2 = T7
EXEC T1 + T7 = T1
EXEC T1 + T1 = T1
EXEC T1 - T5 = T5
EXEC T7 + T5 = T5
```

```
EXEC T7 + T5 = SX
EXEC AM * T2 = T2
EXEC T6 * T6 = T1
EXEC T1 + T2 = T1
EXEC T2 + T2 = T2
EXEC T1 - T2 = T2
EXEC T2 * T2 = T2
EXEC T6 * T1 = T1
EXEC T4 * T7 = T7
EXEC T1 + T7 = T1
EXEC T6 * T7 = T7
EXEC T7 + T7 = T7
EXEC T7 + T7 = T7
EXEC T7 + T7 = T7
EXEC T2 - T7 = QX
EXEC T4 * T1 = T6
EXEC T6 + T6 = T6
EXEC T6 + T6 = QZ
RESTOREPC 0

EXEC DIFFX * QZ = T5          // get QY from
EXEC T5 - QX = T6             //     QX, QZ, SX, SZ
EXEC T6 * T6 = T6
EXEC SX * T6 = T6
EXEC T5 + QX = T5
EXEC DIFFX * QX = T7
EXEC QX * QZ = T1
EXEC AM * QZ = T3
EXEC QZ * QZ = T2
EXEC T3 + T7 = T7
EXEC T5 * T7 = T7
EXEC DIFFY * SZ = T5
EXEC T5 + T5 = T5
EXEC T5 * T2 = QZ
EXEC T5 * T1 = QX
EXEC BM * T2 = T2
EXEC T2 + T2 = T2
EXEC T7 + T2 = T7
EXEC SZ * T7 = T7
EXEC T7 - T6 = QY
RESTOREPC 0
```

```
STOREPC 0 146                    // scalar * P -> Q
JMP 24                           //    2Q -> S
CJMP 150 0 0
STOREPC 0 149
JMP 51                           //    Q <= S + Q,   S <= 2S
JMP 153
STOREPC 0 152
JMP 87                           //    S <= Q + S,   Q <= 2Q
JMP 153
SHIFT
CJMP 146 1
RESTOREPC 1

LOAD 36                          // qz^(O-2) -> T1
EXEC QZ + ZERO = T1
EXEC QZ + ZERO = T2
EXEC T1 * T1 = T1
CJMP 162 0 0
EXEC T1 * T2 = T1
SHIFT
CJMP 159 1
RESTOREPC 0

EXEC SY * QZ = T1                // Q + S => Q
EXEC QY * SZ = T2
EXEC SX * QZ = T3
EXEC QX * SZ = T4
EXEC T1 - T2 = T5
EXEC T3 - T4 = T6
EXEC QZ * SZ = T7
EXEC T6 * T6 = T8
EXEC T8 * T6 = T9
EXEC T2 * T9 = T2
EXEC T8 * T4 = T1
EXEC T5 * T5 = T3
EXEC T3 * T7 = T3
EXEC T3 - T9 = T3
EXEC T3 - T1 = T3
EXEC T3 - T1 = T3
EXEC T3 * T6 = QX
EXEC T1 - T3 = T1
EXEC T5 * T1 = T1
```

```
EXEC T1 - T2 = QY
EXEC T9 * T7 = QZ
RESTOREPC 0

NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP                    // POINT MULTIPLICATION ----------
CP2MEM 2 1                      // copy X from PT
CP2MEM 3 2                      // copy Y from PT
CP2MEM 32 3                     // copy scalar from PT
LOADN
LOAD 32
// load scalar
STOREPC 0 207
JMP 2                           // P(x,y) -> Q(X,Y,Z)
STOREPC 1 209
JMP 144                         // scalar * P -> Q
STOREPC 0 211
JMP 123                         // get QY from
STOREPC 0 213                   //   QX, QZ, SX, SZ
JMP 15                          // qz^(p-2) -> T1
STOREPC 0 215
JMP 10                          // qx, qy, T1(=z^-1)
CP2PT 5 1                       //   -> PX, PY (in T6, T7)
CP2PT 6 2
CP2PT 7 3
CP2PT 16 4
CP2PT 21 5
CP2PT 22 6
END
NOP
NOP                    // ECDSA VERIFICATION ----------
```

```
LOADO
CP2MEM 46 1                    // copy public key from PT
CP2MEM 47 2
CP2MEM 43 3                    // copy r, s and e from PT
CP2MEM 44 4
CP2MEM 45 5
EXEC SIGS * R**2o = QZ         // copy sigS to qz
STOREPC 0 233
JMP 156
// qz^(O-2) -> T1
EXEC HASH * R**2o = T2
EXEC SIGR * R**2o = T3
EXEC T2 * T1 = T7              // u1M_o -> T7
EXEC T3 * T1 = T8              // u2M_o -> T8
EXEC T7 * ONE = T9             // T7 -> u1 (T9)
EXEC T8 * ONE = T10            // T8 -> u2 (T10)
LOADN
LOAD 25                        // load u2
EXEC PUBX + ZERO = PX
EXEC PUBY + ZERO = PY          // load Qa -> P
STOREPC 0 245
JMP 2                          // P(x,y) -> Q(X,Y,Z)
STOREPC 1 247
JMP 144                        // u2 * Qa
STOREPC 0 249
JMP 123                        // get QY from
EXEC QX + ZERO = T11           //   QX, QZ, SX, SZ
EXEC QY + ZERO = T12
EXEC QZ + ZERO = T13
LOAD 24                        // load u1
EXEC GENX + ZERO = PX
EXEC GENY + ZERO = PY          // load G -> P
STOREPC 0 257
JMP 2                          // P(x,y) -> Q(X,Y,Z)
STOREPC 1 259
JMP 144                        // u1 * G
STOREPC 0 261
JMP 123                        // get QY from
EXEC T11 + ZERO = SX           //   QX, QZ, SX, SZ
EXEC T12 + ZERO = SY
EXEC T13 + ZERO = SZ
STOREPC 0 266
```

```
JMP 165                          // Q + S => Q
STOREPC 0 268
JMP 15                           // qz^(p-2) -> T1
STOREPC 0 270
JMP 10                           // qx, qy, T1(=z^-1)
LOADO                            //   -> PX, PY (in T6, T7)
EXEC T6 + ZERO = T2
EXEC T2 - SIGR = T3              // v-r
LOAD 18                          // calculate length of v-r
CJMP 277 2
EXEC ZERO + ZERO = T1
JMP 279
EXEC ONE + ZERO = T1
JMP 279
CP2PT 21 5
CP2PT 22 6
CP2PT 16 7
END
NOP
NOP                    // ECDSA GENERATION ----------------
CP2MEM 32 1                      // copy private key from PT
CP2MEM 45 2                      // copy hash from PT
CP2MEM 51 3
// copy random number from PT
LOADN
LOAD 51
EXEC GENX + ZERO = PX
EXEC GENY + ZERO = PY            // load G -> P
STOREPC 0 294
JMP 2                            // P(x,y) -> Q(X,Y,Z)
STOREPC 1 296
JMP 144                          // rnd * G
STOREPC 0 298
JMP 123                          // get QY from
STOREPC 0 300                    //   QX, QZ, SX, SZ
JMP 15                           // qz^(p-2) -> T1
STOREPC 0 302
JMP 10                           // qx, qy, T1(=z^-1)
LOADO                            //   -> PX, PY (in T6, T7)
EXEC T6 + ZERO = SIGR
EXEC RND * R**2o = QZ            // copy random to qz
STOREPC 0 307
```

```
JMP 156
// qz^(O-2) -> T1
EXEC HASH * R**2o = T2
EXEC SIGR * R**2o = T3
EXEC KEY * R**2o = T4
EXEC T3 * T4 = T5
EXEC T5 + T2 = T8
EXEC T1 * T8 = T9
EXEC ONE * T9 = SIGS
CP2PT 21 4
CP2PT 22 5
CP2PT 43 6
CP2PT 44 7
END
NOP
```

Appendix C

Demonstration

C.1 Introduction

C.1.1 Situating the demonstration

The demonstration illustrates the work of Chapter 4 through 'a reconfigurable audiobox'. This audiobox provides three different filters which each can be bypassed. By using remote reconfiguration, the hardware of these three filters can be updated. Updating a filter will result in an audible result. An overview of the setup is given in Figure C.1.

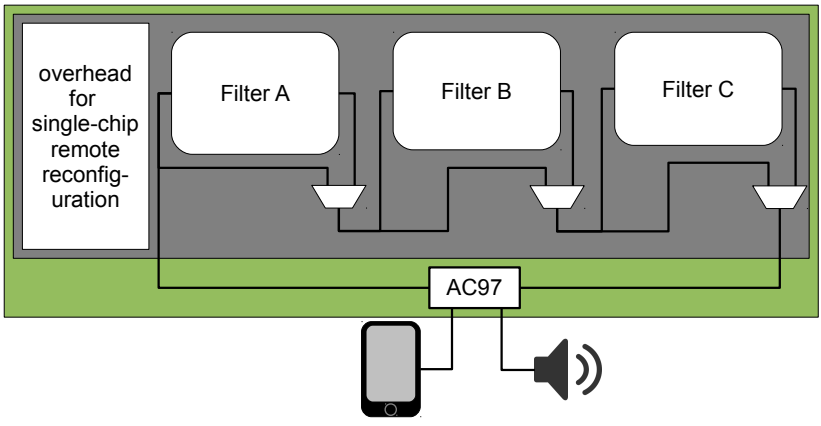

Figure C.1: The setup of the demo

Figure C.1 shows an audio source, which serves as the analogue input for an AC'97 codec chip [29]. The input is converted to a digital value, which is a serial input for the FPGA. The FPGA routes this input to the first filter. The input for the second filter can be chosen, through a multiplexer, to be the input or the output of the first filter. The input for the third filter can, analogously, be chosen between the input or the output of the second filter. Finally, through a third multiplexer, either the input or the output of the third filter is sent out of the FPGA, to the AC'97 codec chip. After conversion of the digital output to an analogue output, the audio signal is played through a speaker. The cryptographic overhead which is required for the single-chip remote reconfiguration is described in Section 4.2.3.

C.1.2 Filters

Three different filters have been generated: an all-pass filter, a low-pass filter, and a high-pass filter. A forth filter-like component which can reside in the reconfigurable partition, is an empty box. Such an empty box is an absence of any connection or configuration in the partition.

the all-pass filter just routes the input signal to the output

the low-pass filter suppresses frequencies in the signal, above 2 547 Hz

the high-pass filter suppresses frequencies in the signal, below 2 547 Hz

C.2 Implementation

C.2.1 Audio interfacing

The ML507 development board holds an AC'97 codec chip. This chip performs an analogue-to-digital conversion on the input and the reverse conversion on the output. Configuring the chip is done through I^2C.

For the demonstration, the sampling frequency is 48 kHz and is binary represented with a 16-bit vector for each channel (left and right). A controlling component configures the AC'97 codec chip on start-up, it provides two 16-bit inputs for the left and right channel, and it sends two 16-bit outputs to the codec chip.

C.2.2 Filters

The empty box filter and the pass-filter are straightforward. Both other filters are implemented as first order, infinite impulse response filters. The low pass filter is implemented as $y_i = \alpha \cdot x_i + (1 - \alpha) \cdot y_{i-1}$ where $\alpha = \frac{\Delta T}{0.25 + \Delta T}$. The high pass filter is implemented as $y_i = \alpha \cdot (x_i - x_{i-1}) + (1 - \alpha) \cdot y_{i-1}$ where $\alpha = \frac{0.5}{0.5 + \Delta T}$. Because the sampling frequency is 48 kHz, ΔT is set to 20833 ns, for both filters.

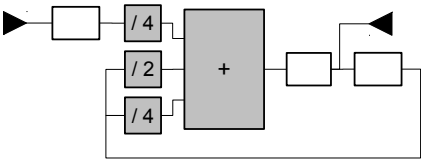

Figure C.2: Low pass filter

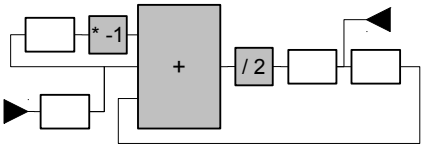

Figure C.3: High pass filter

C.2.3 Generation of the partial bitstreams

In the first step, the bitstreams for the static partition and the filters are generated with Xilinx PlanAhead v14. To provide some flexibility in the filters, three different sizes of reconfigurable areas are implemented. The segmentation of the reconfigurable fabric is shown in Figure C.4 and Table C.1.

Table C.1: Resource availability for each partition

	Slices	BRAMs	DSPs
Filter A	2 886	32	64
Filter B	720	8	16
Filter C	1 440	16	32
Static	6 154	92	16
XC5VFX70T	11 200	148	128

After the final 'bitgen' steps in the toolflow, a large number of bitstreams is available. In the four full bitstreams, the instruction memory needs to be initialised with the program that runs on the MicroBlaze. This is described in Section 3.2.2.

A closer look at the content of the bitstreams shows it contains a header. This header contains metadata amongst which is the file-name of the bitstream. This

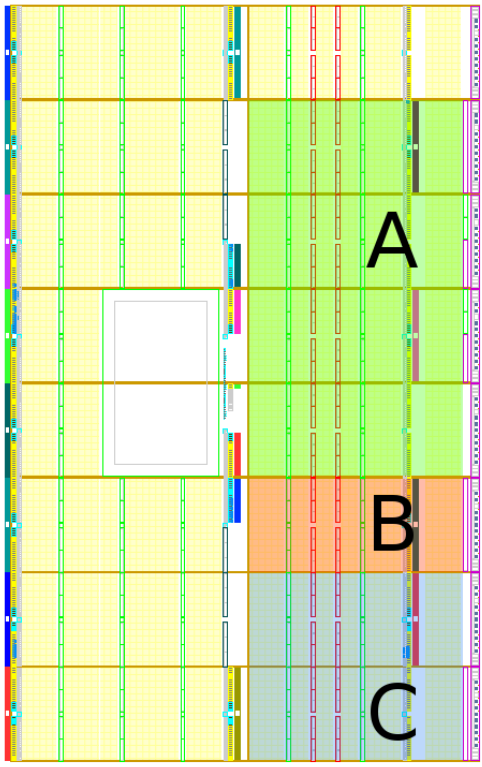

Figure C.4: The segmentation of the floorplan

variable header is the cause of partial bitstream file sizes which are not dividable by data width of the ICAP. Therefore, in a second step, this variable header is discarded. Additionally, the bitstream is pre-padded with 0xFF bytes to make the file size dividable by the ICAP word size.

In the third step, these trimmed partial bitstreams are encoded, using RLE. The result of these operations on the file sizes of the partial bitstreams is shown in Table C.2.

C.2.4 Central Reconfiguration Unit

For the demonstration a standard office-laptop is used as central reconfiguration unit. This laptop is used to generate the bitstreams and has a number of scripts which interact with the FPGA. It locally hosts the visualisation website.

Table C.2: File sizes of the partial bitstreams throughout the generation steps

		bitstream file size after		
		PlanAhead	trimming	RLE
Filter A	empty box	671 986	671 848	24 642
Filter A	all-pass	671 989	671 848	26 264
Filter A	low-pass	671 989	671 848	30 548
Filter A	high-pass	671 989	671 848	32 302
Filter B	empty box	168 490	168 352	7 700
Filter B	all-pass	168 493	168 352	8 576
Filter B	low-pass	168 493	168 352	12 456
Filter B	high-pass	168 493	168 352	16 018
Filter C	empty box	336 322	336 184	6 196
Filter C	all-pass	336 325	336 184	7 206
Filter C	low-pass	336 325	336 184	12 206
Filter C	high-pass	336 325	336 184	14 416

C.2.5 Visualisation

The visualisation of de demonstration is done through a locally hosted websiteThis webpage provides an interface to the different steps of the reconfiguration protocol: the execution of the full STS protocol, the request of a bitstream validation, and the request for a partial reconfiguration.

Two feedback windows are present to report on the actions of the FPGA and the local scripts on the laptop. For the FPGA, this feedback is done through a serial connection which only is available on a demonstration setup.

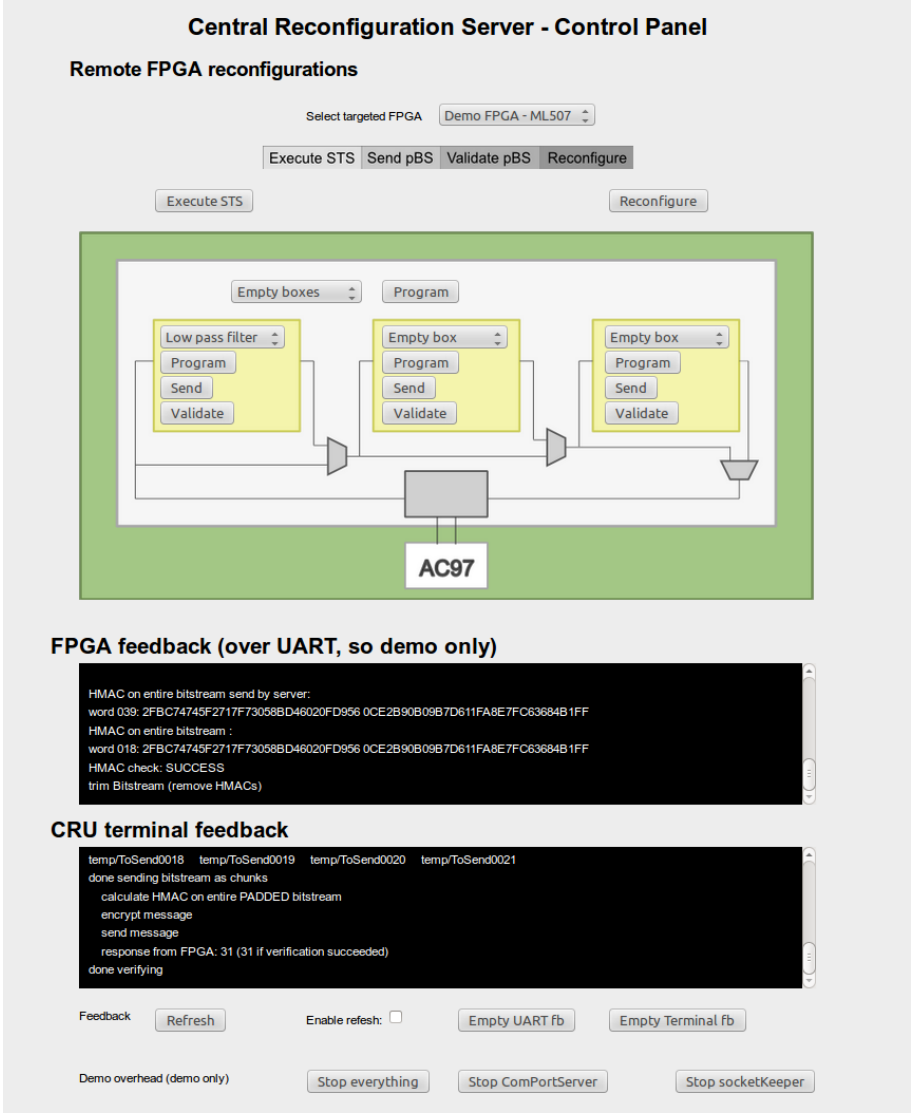

Figure C.5: A screenshot of the web-interface of the demo

Bibliography

[1] Directive 2006/24/EC of the European Parliament and of the Council. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32006L0024:en:HTML, 2006. [Online; accessed 6-March-2012]. pages 54

[2] Beckhoff, C., Koch, D., and Torresen, J. Go Ahead: A Partial Reconfiguration Framework. In *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2012), IEEE, pp. 37–44. pages 102, 108, 113, 115

[3] Bellare, M., and Namprempre, C. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *Journal of Cryptology 21*, 4 (2008), 469–491. pages 95

[4] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. The Keccak SHA-3 submission. http://keccak.noekeon.org/Keccak-submission-3.pdf, 2011. pages 29

[5] Blake-Wilson, S., and Menezes, A. Unknown key-share attacks on the station-to-station (sts) protocol. pages 92

[6] Braeken, A., Genoe, J., Kubera, S., Mentens, N., Touhafi, A., Verbauwhede, I., Verbelen, Y., Vliegen, J., and Wouters, K. Secure remote reconfiguration of an FPGA-based embedded system. In *Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoc)* (2011), Goehringer, Diana and Almeida, Gabriel Marchesan and Sassatelli, Gilles and Indrusiak, Leandro Soares, Ed., IEEE, pp. 74–79. pages 97

[7] Carreira, J., Costa, D., and Silva, J. Fault injection spot-checks computer system dependability. *IEEE Spectrum 36*, 8 (1999), 50–55. pages 91

[8] Castillo, J., Huerta, P., Lopez, V., and Martinez, J. I. A secure self-reconfiguring architecture based on open-source hardware. In *Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig)* (2005), Cumplido, René and Feregrino, Claudia, Ed., IEEE, pp. 7–10. pages 73, 99

[9] Chari, S., Jutla, C. S., Rao, J. R., and Rohatgi, P. Towards sound approaches to counteract power-analysis attacks. In *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO)* (1999), Wiener, Michael J., Ed., vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 398–412. pages 20

[10] Chodowiec, P., and Gaj, K. Very compact FPGA implementation of the AES algorithm. In *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (2003), Walter, Colin D. and Koç, Çetin Kaya and Paar, Christof, Ed., vol. 2779 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 319–333. pages xvi, 48, 49, 52, 59

[11] Cohen, H., Miyaji, A., and Ono, T. Efficient elliptic curve exponentiation using mixed coordinates. In *Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT)* (1998), Springer-Verlag, pp. 51–65. pages xix, 40, 41, 43

[12] Coron, J.-S. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (1999), Koç, Çetin Kaya and Paar, Christof, Ed., vol. 1717 of *Lecture Notes in Computer Science*, Springer, pp. 292–302. pages 39

[13] Ştefan, R., and Coţofană, S. D. Bitstream compression techniques for Virtex 4 FPGAs. In *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL)* (2008), IEEE, pp. 323–328. pages 75, 86

[14] Daemen, J., and Rijmen, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag New York, Inc., 2002. pages 48

[15] Dandalis, A., and Prasanna, V. K. Configuration compression for FPGA-based embedded systems. In *2001 ACM/SIGDA 9th International symposium on Field Programmable Gate Arrays (FPGA)* (2001), ACM, pp. 173–182. pages 75

[16] Devic, F., Torres, L., and Badrignans, B. Secure Protocol Implementation for Remote Bitstream Update Preventing Replay Attacks on FPGA. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL)* (2010), Ferrandi, Fabrizio and Nurmi,Jari and Santambrogio, Marco D., Ed., IEEE, pp. 179–182. pages 74

[17] Devic, F., Torres, L., Crenne, J., Badrignans, B., and Benoît, P. Secure DPR: Secure update preventing replay attacks for dynamic partial reconfiguration. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)* (2012), Koch, Dirk and Singh, Satnam and Tørresen, Jim, Ed., IEEE, pp. 57–62. pages 74, 99

[18] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6 (1976), 644–654. pages 59

[19] Drimer, S., Güneysu, T., Kuhn, M. G., and Paar, C. Protecting multiple cores in a single FPGA design. `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32006L0024:en:HTML`, 2008. pages 102

[20] Drimer, S., and Kuhn, M. G. A Protocol for Secure Remote Updates of FPGA Configurations. In *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC)* (2009), Becker, Jürgen and Woods, Roger and Athanas, Peter M. and Morgan, Fearghal, Ed., vol. 5453 of *Lecture Notes in Computer Science*, Springer, pp. 50–61. pages 73, 99

[21] ECRYPTII. Yearly Report on Algorithms and Keysizes. Tech. rep., KU Leuven, 2012. pages 34, 78

[22] Feller, T., Malipatlolla, S., Meister, D., and Huss, S. A. TinyTPM: A Lightweight Module aimed to IP Protection and Trusted Embedded Platforms. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2011), IEEE Computer Society, pp. 6–11. pages 102

[23] Gassend, B., Clarke, D., van Dijk, M., and Devadas, S. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security (CCS)* (2002), Atluri, Vijayalakshmi, Ed., ACM Press, pp. 148–160. pages 67, 91, 117

[24] Gonzalez, I., Lopez-Buedo, S., and Gomez-Arribas, F. J. Implementation of secure applications in self-reconfigurable systems. *Microprocessors and Microsystems 32*, 1 (2008), 23–32. pages 73

[25] Güneysu, T., Möller, B., and Paar, C. Dynamic Intellectual Property Protection for Reconfigurable Devices. In *International Conference on Field-Programmable Technology (ICFPT)* (2007), Amano, Hideharu and Ye, Andy and Ikenaga, Takeshi, Ed., IEEE, pp. 169–176. pages 102

[26] Haiyun, G., and Shurong, C. Partial Reconfiguration Bitstream Compression for Virtex FPGAs. In *Congress on Image and Signal Processing (CISP)* (2008), vol. 5, IEEE Computer Society, pp. 183–185. pages 75

[27] Hwa Pan, J., Mitra, T., and Wong, W.-F. Configuration Bitstream Compression for Dynamically Reconfigruable FPGAs. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)* (2004), IEEE Computer Society / ACM, pp. 766 – 773. pages 74

[28] Hwang, D., Tiri, K., Hodjat, A., Lai, B.-C., Yang, S., Schaumont, P., and Verbauwhede, I. AES-Based Security Coprocessor IC in 0.18-μm CMOS With Resistance to Differential Power Analysis Side-Channel Attacks. *IEEE Journal of Solid-State Circuits 41*, 4 (2006), 781–792. pages 50

[29] Intel Corporation. High Definition Audio Specification (Revision 1.0a), 2010. pages 136

[30] Information technology - Security techniques - Encryption algorithms - Part 2: Asymmetric ciphers, 2006. pages 59

[31] Izu, T., Möller, B., and Takagi, T. Improved Elliptic Curve Multiplication Methods Resistant against Side Channel Attacks. In *Proceedings of the 3rd International Conference on Cryptology in India (INDOCRYPT)* (2002), Menezes, Alfred and Sarkar, Palash, Ed., vol. 2551 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 296–313. pages xix, 40, 41, 43

[32] Kepa, K., Morgan, F., Kosciuszkiewicz, K., and Surmacz, T. SeReCon: a secure reconfiguration controller for self-reconfigurable systems. *International Journal of Critical Computer-Based Systems 1*, 1/2/3 (2010), 86–103. pages 74, 99

[33] Koblitz, N. Elliptic curve cryptosystems. *Mathematics of Computation 48*, 177 (1987), 203–209. pages 16

[34] Koblitz, N. A Family of Jacobians Suitable for Discrete Log Cryptosystems. In *Proceedings of the 8th Annual International Cryptology Conference (CRYPTO)* (1988), vol. 403 of *Lecture Notes in Computer Science*, Springer, pp. 94–99. pages 19

[35] Koç, Ç. K., Acar, T., and Kaliski, B. S. J. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro 16*, 3 (1996), 26–33. pages 38

[36] Koch, D., Beckhoff, C., and Teich, J. Bitstream Decompression for High Speed FPGA Configuration from Slow Memories. In *International Conference on Field-Programmable Technology (ICFPT)* (2007), Amano, Hideharu and Ye, Andy and Ikenaga, Takeshi, Ed., IEEE, pp. 161–168. pages 75, 86

[37] Koch, D., Beckhoff, C., and Teich, J. ReCoBus-Builder; A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs. In *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL)* (2008), IEEE, pp. 119–124. pages 108

[38] Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO)* (1996), Koblitz, Neal, Ed., vol. 1109 of *Lecture Notes in Computer Science*, Springer, pp. 104–113. pages 19, 74, 92

[39] Kocher, P. C., Jaffe, J., and Jun, B. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO)* (1999), Wiener, Michael J., Ed., vol. 1666 of *Lecture Notes in Computer Science*, Springer, pp. 388–397. pages 20, 74

[40] Krawczyk, H., Bellare, M., and Canetti, R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. pages 29

[41] Maes, R., Schellekens, D., and Verbauwhede, I. A Pay-per-Use Licensing Scheme for Hardware IP Cores in Recent SRAM based FPGAs. *Transactions on Information Forensics and Security 7*, 1 (2012), 98–108. pages xii, xvii, 101, 102, 103, 105, 106, 107, 110, 117

[42] Maes, R., Van Herrewege, A., and Verbauwhede, I. PUFKY: A Fully Functional PUF-based Cryptographic Key Generator. In *Proceedings of the 14th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (2012), Prouff, Emmanuel and Schaumont, Patrick, Ed., vol. 7428 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 302–319. pages 91

[43] Mao, W. *Modern Cryptography: Theory and Practice*. Prentice Hall Professional Technical Reference, 2003. pages 11, 18

[44] McIvor, C., McLoone, M., and McCanny, J. V. An FPGA elliptic curve cryptographic accelerator over GF(p). *IEE Conference Publications 2004* (2004), 589–594. pages 43

[45] Menezes, A. J., Vanstone, S. A., and Van Oorschot, P. C. *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., 1996. pages xv, 11, 46, 75, 77

[46] Mentens, N., Gierlichs, B., and Verbauwhede, I. Power and Fault Analysis Resistance in Hardware through Dynamic Reconfiguration. In *Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (2008), E. Oswald and P. Rohatgi, Eds., vol. 5154 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 346–362. pages 122

[47] Messerges, T. S. Using Second-Order Power Analysis to Attack DPA Resistant Software. In *Proceedings of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (2000), Paar, Christof and Koç, Çetin Kaya, Ed., vol. 1965 of *Lecture Notes in Computer Science*, Springer, pp. 238–251. pages 20

[48] Miller, V. S. Use of Elliptic Curves in Cryptography. In *Proceedings of the 6th Annual International Cryptology Conference (CRYPTO)* (1986), Williams, Hugh C., Ed., vol. 218 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 417–426. pages 16

[49] Montgomery, P. L. Modular Multiplication without Trial Division. *Mathematics of Computation 44*, 170 (1985), 519–521. pages 37, 38

[50] Montgomery, P. L. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation 48*, 177 (1987), 243–264. pages 17

[51] Moradi, A., Kasper, M., and Paar, C. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures - An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism. In *Proceedings of the International Conference on Topics in Cryptology (CT-RSA)* (2012), Dunkelman, Orr, Ed., vol. 7178 of *Lecture Notes in Computer Science*, Springer, pp. 1–18. pages 70, 74, 91, 106, 108

[52] Moradi, A., Poschmann, A., Ling, S., Paar, C., and Wang, H. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Proceedings of the 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)* (2011), Paterson, Kenneth G., Ed., vol. 6632 of *Lecture Notes in Computer*

Science, Springer Berlin Heidelberg, pp. 69–88. pages xvi, 48, 49, 50, 52, 107

[53] Narinx, H., and Ramakers, W. Implementatie en evaluatie van SHA-3-kandidaten op FPGA. Master's thesis, Katholieke Hogeschool Limburg, 2010. promotor: Indesteege, S., and Mentens, N. pages 29, 30

[54] National Institute Of Standards and Technology. Computer Data Authentication (FIPS PUB 113), May 1985. pages 47

[55] National Institute Of Standards and Technology. Specification of the Advanced Encryption Standard (FIPS PUB 197), November 2001. pages 20, 48

[56] National Institute Of Standards and Technology. The keyed-hash message authentication code (FIPS PUB 198-1), July 2008. pages 29

[57] National Institute Of Standards and Technology. Recommendation for Applications using Approved Hash Algorithms (SP800-107), August 2012. pages 93

[58] National Institute Of Standards and Technology. Recommendation for Block Cipher Modes of Operation (SP800-38A), March 2012. pages 46

[59] National Institute Of Standards and Technology. Secure Hash Standard (FIPS PUB 180-4), March 2012. pages 29, 52, 80

[60] National Institute Of Standards and Technology. Digital Signature Standard (FIPS PUB 186-4), July 2013. pages xvi, 19, 34, 78, 80

[61] Örs, S. B., Batina, L., Preneel, B., and Vandewalle, J. Hardware Implementation of an Elliptic Curve Processor over GF(p). In *Proceedings of the 14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)* (2002), IEEE, pp. 433–443. pages 43

[62] Osborne, A. *An Introduction to Microcomputers: Basic concepts*, 2nd ed. McGraw-Hill, 1980. pages 111

[63] Pedroni, V. A. *Digital Electronics and Design with VHDL*. Morgan Kaufmann, 2008. pages 10

[64] Pulls, T., Wouters, K., Vliegen, J., and Grahn, C. Distributed Privacy-Preserving Log Trails. Karlstad University Studies 2012:24, Karlstad University, 2012. pages 53, 54, 55, 58, 68

[65] Qin, X., Muthry, C., and Mishra, P. Decoding-Aware Compression of FPGA Bitstreams. *Transactions on Very Large Scale Integration (VLSI) Systems 19*, 3 (2011), 411–419. pages 75

[66] Sakiyama, K., Mentens, N., Batina, L., Preneel, B., and Verbauwhede, I. Reconfigurable modular arithmetic logic unit supporting high-performance RSA and ECC over GF(p). *International Journal of Electronics 94* (2007), 501–514. pages 43

[67] Schulz, R. A. Random number generator circuit (patent US 4905176 A), Feb 1990. pages 32

[68] Shoup, V. A Proposal for an ISO Standard for Public Key Encryption. *IACR Cryptology ePrint Archive 2001* (2001), 112. pages 59

[69] Simpson, E., and Schaumont, P. Offline Hardware/Software Authentication for Reconfigurable Platforms. In *Proceedings of the 8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (2006), Goubin, Louis and Matsui, Mitsuru, Ed., vol. 4249 of *Lecture Notes in Computer Science*, Springer, pp. 311–323. pages 102

[70] Sunar, B., Martin, W., and Stinson, D. A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks. *Transactions on Computers 56*, 1 (2007), 109–119. pages 32

[71] Ünal, Ö., and Sallaerts, B. Optimalisatie van een trusted state voor een gedistribueerd logging concept. Master's thesis, Katholieke Hogeschool Limburg, 2013. promotor: Wouters, Karel and Mentens, Nele. pages 49

[72] Varchola, M., Güneysu, T., and Mischke, O. MicroECC: A Lightweight Reconfigurable Elliptic Curve Crypto-processor. In *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)* (2011), Athanas, Peter M. and Becker, Jürgen and Cumplido, René, Ed., IEEE Computer Society, pp. 204–210. pages 43

[73] Vliegen, J., Mentens, N., Genoe, J., Braeken, A., Kubera, S., Touhafi, A., and Verbauwhede, I. A compact FPGA-based architecture for elliptic curve cryptography over prime fields. In *Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2010), Wolinski, Christophe and Teich, Jürgen and Charot, François and Najjar, Walid, Ed., IEEE, pp. 313–316. pages 43, 51

[74] Vliegen, J., Mentens, N., and Verbauwhede, I. A Single-chip Solution for the Secure Remote Configuration of FPGAs using Bitstream Compression. In *Proceedings of the 2013 International Conference on*

Reconfigurable Computing and FPGAs (ReConFig) (2013), Cumplido, René and De la Torre, Eduardo and Wirthlin, Mike, Ed., IEEE Computer Society, p. 6. pages 97

[75] Vliegen, J., Wouters, K., Grahn, C., and Pulls, T. Hardware strengthening a Distributed Logging Scheme. In *Proceedings of the 15th Euromicro Conference on Digital System Design (DSD)* (2012), IEEE Computer Society, IEEE, pp. 171–176. pages xi, 53, 68

[76] Walter, C. D. Montgomery exponentiation needs no final subtraction. *Electronic letters 35* (1999), 1831–1832. pages 38

[77] Wold, K., and Tan, C. H. Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings. *International Journal of Reconfigurable Computing 2009*, 501672 (2009), 385–390. pages 32, 33, 52

[78] Xilinx. Virtex-5 Family Overview (DS100), February 2009. pages 8

[79] Xilinx. Spartan-6 Family Overview (DS160), March 2010. pages 8

[80] Xilinx. Xilinx Implementation Strategies using FPGA Editor, 2010. pages 110

[81] Xilinx. Xilinx Partial Reconfiguration User Guide (UG702), 2010. pages 108

[82] Xilinx. 7 Series FPGAs Configuration (UG470), 2013. pages 7

[83] Xilinx. Virtex-6 FPGA Configuration (UG360), 2013. pages 7

[84] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *Transactions on Information Theory 23*, 3 (1977), 337–343. pages 86

List of publications

Refereed Articles

2011 Verbelen, Y., Braeken, A., Kubera, S., Mentens, N., Touhafi, A., and Vliegen, J. Implementation of a Server Architecture for Secure Reconfiguration of Embedded Systems. In *ARPN Journal of Systems and Software*, vol. 1, no. 9, pp. 270-279.

Refereed Articles to Appear

2014 Vliegen, J., Mentens, N., and Verbauwhede, I. Secure, remote, dynamic reconfiguration of FPGAs. In *ACM Transactions on Reconfigurable Technology and Systems*.

Refereed Articles in revision

2014 Vliegen, J., Mentens, N., Koch, D., Schellekens, D., and Verbauwhede, I. Practical Feasibility Evaluation and Improvement of a Pay-per-Use Licensing Scheme for Hardware IP Cores in FPGAs. In *Journal of Cryptographic Engineering*.

International Conference Papers

2013 Vliegen, J., Mentens, N., and Verbauwhede, I. A Single-chip Solution for the Secure Remote Configuration of FPGAs using Bitstream Compression. In *Proceedings of the 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, IEEE Computer Society, pp. 6.

2012 Vliegen, J., Wouters, K., Grahn, C., and Pulls, T. Hardware strengthening a Distributed Logging Scheme. In *Proceedings of the 15th Euromicro Conference on Digital System Design (DSD)*, IEEE, pp. 171-176.

2011 Braeken, A., Genoe, J., Kubera, S., Mentens, N., Touhafi, A., Verbauwhede, I., Verbelen, Y., Vliegen, J., and Wouters, K. Secure remote reconfiguration of an FPGA-based embedded system. In *Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoc)*, IEEE, pp. 74-79.

2010 Batina, L., Hogenboom, J., Mentens, N., Moelans, J., and Vliegen, J. Side-channel evaluation of FPGA implementations of binary Edwards curves. In *Proceedings of the 2010 International Conference on Electronics, Circuits, and Systems (ICECS)*, IEEE, pp. 1255-1258.

2010 Vliegen, J., Mentens, N., Genoe, J., Braeken, A., Kubera, S., Touhafi, A., and Verbauwhede, I. A compact FPGA-based architecture for elliptic curve cryptography over prime fields. In *Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, pp. 313-316.

2009 Braeken A., Kubera S., Trouillez F., Touhafi A., Vliegen J., and Mentens, N. Secure FPGA technologies and techniques. In *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, pp. 560-563.

Workshop Papers

2013 Vandorpe, J., Bartkewitz, T., Drutarovsky, M., Hafting, Y., Koch, D., Lemke-Rust, K., Mentens, N., Plöger, P., Samarin, P., Smeets, R., Tørresen, J., Varchola, M., and Vliegen, J. Remote FPGA Design Through eDiViDe - European Digital Virtual Design Lab. In *23rd International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, pp. 2.

2013 Vandorpe, J., Bartkewitz, T., Drutarovsky, M., Hafting, Y., Koch, D., Lemke-Rust, K., Mentens, N., Plöger, P., Samarin, P., Smeets, R., Tørresen, J., Varchola, M., and Vliegen, J. eDiViDe: European Digital Virtual Design Lab - A Remote Learning Platform for Digital Design. In *Proceedings of the SEFI Annual Conference*, SEFI & KU Leuven, pp. 8.

Other

2012 Pulls, T., Wouters, K., Grahn, C., and Vliegen, J. Distributed Privacy-Preserving Log Trails. In *Technical report 2012:24*, Dept. of Computer Science, Karlstad University, Sweden, pp. 147.

2010 Mentens, N., Vliegen, J., Braeken, A., Touhafi, A., Verbauwhede, I., and Wouters, K. Secure remote reconfiguration of FPGAs. In *Dynamically Reconfigurable Architectures*, IBFI, no. 10281, pp. 4.

Master Theses supervised

2013 Awouters, S., and Rekoms, B. Digitale signaalverwerking op FPGA met behulp van partiële herconfiguratie. Master Thesis, KHLim dept. IWT, 58 pages.

2013 Sallaerts, B., and Ünal, Ö. Optimalisatie van een trusted state voor een gedistribueerd logging concept. Master Thesis, KHLim dept. IWT, 47 pages.

2012 Smeets, R., and Volders, J. Het implementeren van een meerkanaals gnss-ontvanger met flexibele herconfiguratie op een FPGA. Master Thesis, KHLim dept. IWT, 97 pages.

2012 Schreurs, B. Ontwerp van een firewire-interface op FPGA. Master Thesis, KHLim dept. IWT, 62 pages.

2012 Dewallef, B., and Jacobs, L. Ontwerp van een Wi-Fi to Wireless Sensor Network bridge op FPGA. Master Thesis, KHLim dept. IWT, 96 pages.

2010 Vandeweerdt, R., and Grondelaers, J. Compacte implementatie voor een veilige FPGA-configuratie via een ethernetverbinding. Master Thesis, KHLim dept. IWT, 58 pages.

2010 Ramakers, W., and Narinx, H. Implementatie en evaluatie van SHA-3-kandidaten op FPGA. Master Thesis, KHLim dept. IWT, 81 pages.

2010 Moelans, J., and Swinnen, S. Implementatie van vermogenaanvallen en tegenmaatregelen op FPGA. Master Thesis, KHLim dept. IWT, 162 pages.

2010 Aerts, R., and Sluismans, B. Efficiënte implementatie van cryptosystemen gebaseerd op elliptische krommen. Master Thesis, KHLim dept. IWT, 79 pages.

FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
COSIC
Kasteelpark Arenberg 10, bus 2452
B-3001 Leuven
jo.vliegen@esat.kuleuven.be
http://www.esat.kuleuven.be